INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

NOTE TO USERS

The original manuscript received by UMI contains pages with slanted print. Pages were microfilmed as received.

This reproduction is the best copy available

UMI

**A STUDY IN SOFTWARE REUSE:
DESIGN, IMPLEMENTATION, AND
MEASUREMENTS
ON A
DISTRIBUTED CLIENT/SERVER
SYSTEM**

by

Leslie J. Steele

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Information and System Science

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University,
Ottawa, Ontario, Canada

July 1997

copyright
1997, Leslie J. Steele

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-27060-2

The undersigned hereby recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

*A Study in Software Reuse:
Design, Implementation, and Measurements
on a
Distributed Client/Server System*

submitted by Leslie J. Steele, B.Sc.
in partial fulfillment of the requirements
for the degree of
Master of Information and System Science

Dr. D. C. Petriu, Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
June , 1997

Abstract

Software reuse increasingly emerges as an issue of importance as the mean time-to-market for application software shrinks. The creation of reliable, extensible, and maintainable software that can easily be reused in new applications, thus decreasing development time, is highly desirable. This thesis explores some of the difficulties and issues surrounding software reuse via a case study. A small performance analysis of the OO design implemented in the case study is included as a basis from which further studies into the performance of OO systems can devolve.

The case study presented in this thesis consists of the design, implementation, and performance measurements on a client/server system comprised of multiple clients and a remote database server. The system reuses two existing components: (1) a database system designed for a single interactive user on a single node and (2) the ACE framework which links the clients and the remote server by providing distribution and concurrency. Design patterns are employed to abstract the design of the new system and the benefits of OO technology are explored as pertains to software reuse. Average system throughput and response time are measured for varying numbers of threads in the multi-threaded server, as well as CPU service time in the database server.

*To my children, Greer and Stewart,
for the love, joy, and light
they have brought me*

*To Cheryl & Ken Creech
and Dawn Bartsch,
for simply always being there*

Acknowledgements

I would like to express my deepest thanks to my supervisor, Professor Dorina Petriu, for her valuable advice and guidance. Her perception, patience, and understanding are of a rare quality. She has my deepest gratitude for making this thesis possible.

I would also like to express thanks and best wishes to my colleagues in the Real-Time and Distributed Systems Lab.

To

Cheryl Schramm, for her encouragement, assistance, and care both technical and personal;

Istabrak Abdul-Fatah, for being a brother;

Gurudas Somadder, for providing friendship and technical assistance. Guru's thread pool code is reused in this thesis and the skeleton for the experiment script is based on his work;

Alex Hubbard, for countless times of assistance above and beyond the call of duty;

Fahim Sheikh, for his encouragement and friendship;

Marc Courtois, Andrew Miga, Mona Metea, Tunde Adebayo, Peter Kwong, and Nigel Cook, for their friendship.

I would like to express my gratitude to my professors, Profs. Pearce, Lambadaris, Karam, Woodside, Ritcher, Neilson, and Lee. Their courses were deeply enriching.

Table of Contents

Abstract	ii
Acknowledgements	iv
Table of Contents	v
List of Figures	x
List of Tables	xii
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Content Outline	3
2. BACKGROUND	5
2.1 Object-Oriented System Concepts	5
2.2 Design Patterns	7
2.3 Databases	9
2.4 Distributed Systems	13
2.4.1 DCE	14
2.4.2 CORBA	15
2.5 Performance Measurements	17
3. THE ACE TOOLKIT	20
3.1 Structure of the ACE Toolkit	20
3.2 Design Patterns Implemented by ACE	22
3.2.1 Reactor Design Pattern	23
3.2.1.1 Intent	23

3.2.1.2	Motivation	23
3.2.1.3	Applicability	24
3.2.1.4	Structure	24
3.2.1.5	Participants	24
3.2.1.6	Collaborations	26
3.2.2	Acceptor Pattern.	26
3.2.2.1	Intent	26
3.2.2.2	Motivation	26
3.2.2.3	Applicability	28
3.2.2.4	Structure and Participants	29
3.2.2.5	Collaborations	29
3.2.3	Connector Pattern	31
3.2.3.1	Intent	31
3.2.3.2	Motivation	31
3.2.3.4	Applicability	32
3.2.3.5	Structure and Participants	32
3.2.3.6	Collaborations	32
3.2.4	Active Object Pattern	35
3.2.4.1	Intent	35
3.2.4.2	Motivation	35
3.2.4.3	Applicability	37
3.2.4.4	Structure and Participants	37
3.2.5	Lightweight OS Concurrency Mechanisms	39

4.	PARODY OBJECT DATABASE	42
4.1	PARODY Database Definition/Structure	43
4.1.1	Persistent Object Class Definition	43
4.1.2	Keys	47
4.1.3	Class Relationships	48
4.1.4	Multiple-Copy Objects	49
4.1.5	PARODY File Architecture	49
4.1.5.1	Nodes	50
4.1.5.2	Object File	52
4.1.5.3	Index File	52
5.	SYSTEM DESIGN	53
5.1	System Description	53
5.1.1	System Mandate	53
5.1.2	Assumptions	54
5.1.3	Issues	55
5.1.3.1	System Architecture	55
5.1.3.2	Communication Management	56
5.1.3.3	Concurrency Management	56
5.2	System Architecture	56
5.2.2	Client Architecture	57
5.2.2.1	Transaction Management Code (TMC)	57
5.2.2.2	Proxy Pattern	63
5.2.2.3	Connector Pattern	66

5.2.3	Server Architecture	66
5.2.3.1	Acceptor and Reactor	74
5.2.3.2	Active Object	75
5.2.3.3	Concurrency Mechanism	77
5.3	User Involvement in the EODMS	78
5.3.1	User Involvement in Client Part of EODMS	80
5.3.1.1	Header Files	81
5.3.1.2	Application-Specific Classes	82
5.3.1.3	Transaction Management Code (TMC)	82
5.3.2	User Involvement in the Server Part of EODMS	83
5.3.2.1	Header Files	83
5.3.2.2	Application-Specific Derived Persistent Classes	84
6.	PERFORMANCE MEASUREMENTS AND RESULTS.	86
6.1	Class Presentation and Set-Up	87
6.1.1	Employee class.	88
6.1.2	Department class	89
6.1.3	Project class	90
6.1.4	Assignment class	90
6.2	Collection of Performance Measures	91
6.2.1	System Throughput	91
6.2.2	Server Service Time Characterization	93
6.3	Results and Analysis.	93
6.3.1	System Throughput for Write_Lock Case	93

6.3.2	System Throughput for Read/Write_Lock Case.	95
6.3.3	Service Time Characterization of EODMS Server	98
7.	CONCLUSIONS	103
7.1	Difficulties in Software Reuse	103
7.1.1	Impedance Mismatch	103
7.1.2	Differing Internal Assumptions in Components	104
7.1.3	Differing Internal Assumptions of Control	104
7.2	Overcoming Difficulties in Software Reuse	105
7.2.1	Benefits of OO Technology	105
7.2.2	Benefits of Design Patterns	105
7.3	Performance Measurements	106
7.4	Future Work	106
	REFERENCES	108
	APPENDIX A (User-Supplied Header Files (EODMS Client))	111
	APPENDIX B (User-Supplied Header Files (EOMS Server))	113
	APPENDIX C (Batch Script for Performance Experiments)	116
	C.1 Operation of Main Batch Script	116
	C.1.1 Initialization Section	118
	C.1.2 Experiment Loop Section	118
	C.1.3 Interrupt Controller	122
	C.2 Operation of Experiment File	122

List of Figures

Figure 2.1	Differences Between Class Libraries and Frameworks	10
Figure 2.2	Transient vs.Persistent Objects	12
Figure 2.3	Basic Services and Facilities Provided by DCE	16
Figure 2.4	Basic CORBA Communication Architecture	18
Figure 3.1	Components of ACE Toolkit	21
Figure 3.2	Class Diagram for Reactor Pattern	25
Figure 3.3	Message Sequence Diagram for Reactor Pattern	27
Figure 3.4	Message Sequence Diagram for Acceptor Pattern	30
Figure 3.5	Primary Participants in Connector Pattern	33
Figure 3.6	Message Sequence Diagram for Connector Pattern (Asynchronous Connections)	34
Figure 3.7	Message Sequence Diagram for Connector Pattern (Synchronous Connections)	36
Figure 3.8	Structure of Major Participants in Active Object Pattern	38
Figure 4.1(a)	Flow of Method Calls to Input Persistent Object	45
Figure 4.1(b)	Flow of Method Calls to Save Persistent Object	45
Figure 4.2	Constructor of Persistent Object in PARODY ODMS	46
Figure 4.3	PARODY ODMS File Header and Node Structure	51
Figure 5.1	General Activities of EODMS Client and Server	58
Figure 5.2	EODMS Client Architecture from Design Pattern Perspective	59
Figure 5.3	Class Diagram for EODMS Client	60
Figure 5.4	Object Diagram for EODMS Client	61

Figure 5.5	Message Sequence Chart for EODMS Client (Retrieve Operation)	62
Figure 5.6	Message Sequence Chart for Proxy Pattern	65
Figure 5.7	EODMS Server Architecture from Design Pattern Perspective	67
Figure 5.8	Object Diagram for EODMS Server	68
Figure 5.9	Class Diagram for EODMS Server	69
Figure 5.10(a)	Message Sequence Chart for EODMS Server (Connection Initialization Phase)	70
Figure 5.10(b)	Message Sequence Chart for EODMS Server (Service Initialization Phase)	71
Figure 5.10(c)	Message Sequence Chart for EODMS Server (Service Processing Phase)	72
Figure 5.10(d)	Message Sequence Chart for EODMS Server (Service Completion and Server Shutdown Phase)	73
Figure 5.11	EODMS Server Architecture	76
Figure 6.1	System Throughput vs. Number of Clients (Write_Lock on Retrieve and Change Operations)	94
Figure 6.2	System Throughput vs. Number of Clients (Read_Lock on Retrieve/Write_Lock on Change)	96
Figure 6.3	Response Time vs. Number of Clients (Read_Lock on Retrieve/Write_Lock on Change)	97
Figure 6.4	CPU Service Time Characterization for EODMS Server (Retrieve Operation)	100
Figure 6.5	CPU Service Time Characterization for EODMS Server (Change Operation)	101

List of Tables

Table 1: Service Time Characterization of EODMS Server for Retrieve and Change Operations.	99
--	----

CHAPTER 1

INTRODUCTION

1.1 Motivation

As software systems become increasingly complex and the mean-time-to-market for the delivery of these systems decreases, designers are caught in the bind of trying to produce quality software in a shorter and shorter period of time. Consumer expectations for user-friendly and efficient software, as well as systems which possess all the "bells and whistles" to which users have become accustomed, can place inordinate demands on software designers who are often learning an advanced technology and implementing a corresponding design almost simultaneously. The only feasible way in which complicated yet reliable code can be generated within a reasonable development time and meet user expectations in quality, is to reuse successful design solutions to the software issues involved in the particular application.

Although reuse of successful solutions to known software problems appears to be a desirable and efficient track on which to proceed when developing new software systems, this path has pitfalls and obstacles which must be understood and appreciated or the unwary designer may find himself/herself facing difficulties which had not been anticipated.

The object-oriented (OO) paradigm offers the potential of reusability, reliability,

maintainability, and extensibility of software via the properties of encapsulation, inheritance, and polymorphism [1]. Objects, comprised of both data members and operations on those data (methods), can prevent inadvertent access to their data members. Only via a well-defined interface is access to an object's data members allowed. This information hiding promotes maintainability of code as an object's data members cannot usually be inadvertently changed by users. Objects, as instances of classes, have well-defined interactions that occur between themselves. This allows for greater ease in visualizing and understanding the interplay between objects. Increased clarification of object interaction leads to increased understanding of the software system which enhances the possibility of reuse.

This potential for reuse, however, is only realized when an OO system is designed with flexibility and possible extension in mind. If object relationships are rigid and hard to test and change, reuse becomes difficult.

Design patterns are abstractions which describe successful solutions to common recurring software problems [2]. The purpose of design patterns is to capture experience in such an abstract manner that the reasoning behind what the pattern does and why it does it, is clear. Patterns enable developers to communicate architectural knowledge which assists other developers in learning a new design or architectural style, thus communicating knowledge which prevents unnecessary and costly learning experiences.

The benefits of OO technology and design patterns promise to ease the software designer's burden by facilitating software reuse, but these areas are still relatively new and continue to be explored. Questions arise as to how OO software can best be made

reusable; under what conditions can maximum reusability be achieved; and, importantly, how does incorporating the OO design paradigm into software affect its performance.

This thesis seeks to explore the above issues. The objectives of this thesis are to:

- identify and explore the practical difficulties of OO software reuse via a case study which reuses original components when incorporating distribution and concurrency into a monolithic database application
- design and implement the new distributed system using design patterns and the OO paradigm
- perform performance measurements on the distributed software system in order to provide a basis for building and validating a Layered Queuing Network (LQN) model of the same system as a means of exploring performance in OO systems. Note that the LQN modeling has not been conducted as part of this thesis and is left for future work.

1.2 Content Outline

Chapter 2 gives background information and key concepts relevant to this thesis. Areas covered are OO system concepts, design patterns, relational and object databases, distributed system concepts, and software system performance issues.

Chapter 3 presents an introduction to the ACE framework [3], a key reused component which provides the distribution and communication in the software system designed and implemented in this thesis. The essential design patterns reified by the ACE framework are also discussed.

Chapter 4 provides an overview of the PARODY database, the monolithic database application which comprises another of the key components reused in the development of the new distributed system. Its basic architecture and behaviour are described.

Chapter 5 explains the process involved in the design of the distributed system by reusing the ACE framework and the PARODY database system. The use of design patterns in achieving the new design is presented as is the architecture of the new system.

Chapter 6 presents a limited performance study conducted on the distributed system. System throughput for two different locking strategies and varying number of threads in the server thread pool is provided. Server CPU consumption for two of the database operations is also given.

Chapter 7 summarizes conclusions and suggests future work.

CHAPTER 2

BACKGROUND

This chapter summarizes the major issues and concepts relevant to this thesis. A discussion of key OO concepts occurs in the first section followed by an overview of the principles of design patterns. Relational and OO databases are presented in the third section followed by a brief introduction to the concept of distribution in computer systems. This chapter concludes with a brief examination of some of the issues involved in obtaining performance measurements.

2.1 Object-Oriented System Concepts

Although the principles of structured programming improved the clarity, reliability, and ease of maintenance of software, large-scale software development and implementation remain an often formidable challenge. As applications grow more complex and the mean-time-to-market shrinks, reusability of code becomes a more urgent issue. OO software design brings a new approach to these challenges. Object technology seeks to design and implement software in well-defined modular components which interact with each other to provide some desired functionality.

Applications are not organized into data values and procedures that are separate from one another. Problem domains are dealt with by concentrating on the objects and allowing the characteristics of those objects to determine the procedures to apply to

them. This is the approach of the OO software design paradigm. Various key terms arise in this paradigm which will be presented and briefly discussed in the following paragraphs.

An *object* is an abstraction which represents a concrete entity, with its behavior and interactions. An object has a data structure and performs a set of actions or operations. Objects interact with each other by sending, receiving, and processing messages. An object is asked to perform one of its operations by sending it a message. The implementation of these operations (or methods) need not be known by any other requesting component [1, 4, 5].

The message passing between objects may be either synchronous or asynchronous. During synchronous message passing, the object which sends the message blocks and waits until the receiving object obtains the message, processes it, and replies to the sender object. Asynchronous message passing, however, does not see the sending object halt execution after sending the message. The sender and receiver objects execute concurrently since the sender does not wait for a reply from the receiver before continuing.

Encapsulation is the means by which an object prevents its data members from being incorrectly accessed. An object groups and hides its private data members only allowing other objects to access its data via a well-defined public interface. The methods of the object form this interface.

A *class* defines the data members and behaviors for a group of objects. A class is a

generic specification or template for an object and every object is an instance of a particular class. Any number of objects of a specific class may be instantiated.

Polymorphism is a term which means "the ability to take different forms". In the OO paradigm, polymorphism is the ability of different kinds of objects to respond to the same message. The same method name can have quite different functionality depending on the implementation of the method in the type of receiving object. Thus numerous classes may each have a method with a common name. The object invoking the method name may not know the class of the object to which it is sending a message, however, it is the responsibility of the receiving object to provide the correct implementation of the invoked method. The property of polymorphism in OO systems provides a more straightforward interface for invoking methods and reduces code.

Inheritance is a property of OO systems in which a class can acquire the characteristics of another class. The original class is called the base class, and the new class is called the derived class. The data representation and behavior of the base class is inherited by the derived class except where the derived class modifies selected behavior by overloading specific methods. A derived class may descend from a single base class (single inheritance) or from more than one base class (multiple inheritance). Inheritance provides increased opportunity for software reuse by capitalizing on data and behavior, resident in the base class, which already exists.

2.2 Design Patterns

The emergence of OO technology has assisted designers in reusing code, however, the

development of efficient, robust, extensible, and reusable software is difficult to achieve though this technology alone. The principles of OO technology are application-independent and thus provide, in a non-contextual fashion, ways of reusing code. Many successful solutions to problems that have arisen while developing software exist within a particular context. However, the majority of these solutions reside within the minds of expert developers or are buried deep within source code. Software reuse is raised to a new plane when not only code reuse is present but design reuse is possible and accessible. Design patterns and frameworks help to capture, articulate, and instantiate these successful techniques.

Design patterns represent solutions to problems that arise when developing software within a particular context. Static and dynamic structure and collaboration between key participants in a design are captured by the associated design pattern. The resolution of non-functional forces, such as reusability, modularity, and extensibility, is articulated via design patterns thus they promote reuse of successful software architectures and designs.

Reuse of the architectural and design knowledge provided by design patterns is not enough, however. To reuse code, developers must also reuse detailed designs, algorithms, interfaces, and implementations. Frameworks are the means by which this goal is achieved [2].

A framework is "an integrated collection of components that collaborates to produce a reusable architecture for a family of related applications". The framework provides the reification of the abstract design knowledge contained within the associated design pattern. Although there may seem to be a strong resemblance between frameworks and

conventional class libraries, there are distinct differences. Frameworks are semi-complete applications which address a particular application domain. As Figure 2.1 shows, frameworks provide a reverse direction of control from conventional class libraries. In general, context-specific applications are produced by inheriting from and instantiating framework components [2].

2.3 Databases

The object database is a reasonably new concept, although its precursor, the relational database, evolved in the 1970s and is a proven technology [4]. Relational technology grew as problems (ie. embedded file pointers) associated with traditional hierarchical and network databases needed to be solved.

The relational data model is based on the concept of a *relation*, which physically is represented as a table. These tables are used to contain information about the data entities to be represented in the database. A relation is represented as a two-dimensional table containing rows and columns. Individual records correspond to the rows, or *tuples*, of the table while attributes correspond to the table columns. For instance, the DOG relation is represented by the DOG table. This table has columns for attributes NAME, BREED, OWNER, and COLOR. Allowable values for an attribute constitute the *domain* of that attribute. Two or more attributes may have the same domain or each attribute may have its own distinct domain [6]. A table representing a relation has the following characteristics:

- only one value may reside in each table cell
- each table column has a unique name

***CLASS LIBRARIES
VS.
OBJECT-ORIENTED FRAMEWORK***

Figure 2.1
Differences Between Class Libraries (top) and
Frameworks (bottom)

- values in column cells come from the same domain
- each row (tuple) is unique
- order of rows and columns is immaterial

It is always possible to identify a table row uniquely since no duplicate tuples are permissible. This means that a relation always has a *primary key*, an attribute whose value uniquely identifies the tuple. *Foreign keys* may also be included. A foreign key is an attribute or combination of attributes of a relation that is a primary key of another relation [6]. An integrity rule for foreign keys states that if a foreign key exists in a relation then either the foreign key must match the primary key value of some tuple in the relation for which it is the primary key, or the foreign key value must be null.

As OO applications increasingly become constructed as sequences of programs and execute on multiple machines in a network, designers require that objects exist for a period longer than the execution of the program in which they were created. Such an object is said to be persistent as opposed to a transient object which does not exist past that program's execution [5]. Figure 2.2 illustrates the difference between transient and persistent objects. As shown, persistent objects must be allocated space on physical storage by an Object Database Management System (ODMS) at runtime to ensure their continuation after program execution.

As OO technology began to emerge, limitations in relational database technology to support this new programming paradigm started to surface. References in objects, embedded objects, virtual functions, and pointers in objects are all information pertinent to an object which must be preserved with it. Relational databases could not support the

Transient Objects

Non-volatile storage (Disk)

Persistent Objects

Figure 2.2
Transient vs. Persistent Objects

above features for object persistence thus the object database was conceived. Different formulations exist, however, object databases have the following objectives [4]:

- maintenance of objects: database provides methods for an application to delete and change objects that exist in the database.
- object integrity: database does not store ambiguous objects or appear to retrieve objects that have not previously been stored.
- class relationships: database maintains relationships that exist between application classes
- copies of objects: database is aware of multiple memory copies of a persistent object

2.4 Distributed Systems

The past two decades have seen an explosion in the existence of LANs, WANs, and distributed computing systems in general. The sharing of information, resources, and analysis made possible by a distributed system magnifies the capability and value of both the people and technology associated with the network. But these benefits came at a cost. Increased system complexity introduced by the communication between nodes in the network, as well as by the mixture of applications running on a variety of hardware and software configurations, occurred. The need for a coherent and seamless environment in which distributed applications could run has resulted in numerous such environments, two major players of which are the Distributed Computing Environment (DCE), and more lately, the Common Object Request Broker Architecture (CORBA). The ACE framework for distribution and communication in event-driven applications is much smaller, simpler, and has less overhead than DCE and CORBA. A brief description of

the DCE and CORBA midwares follows while ACE will be presented in Chapter 3.

2.4.1 DCE

DCE, created by the Open System Foundation (OSF), is a distributed computing environment which is built on the top of existing operating systems such as UNIX, VMS, WINDOWS, and OS/2. The primary goal of DCE is to provide a platform for executing distributed applications in a manner which is largely transparent to the user [7]. A DCE application can run on a distributed system which is heterogeneous, comprised of computers from various vendors each of which is running its own local operating system. The DCE software provides a layer on top of the operating system hiding the differences and automatically doing necessary conversions. None of this is visible to the application programmer.

A large number of tools and services are provided in the DCE package plus the infrastructure in which they operate. The client/server model is the underlying paradigm of DCE. Clients (user processes) access remote services offered by server processes. Application programmers provide the components related to the application while other services are part of DCE itself. DCE offers the following distributed services and facilities for distributed programming [7]:

- distributed time service: performs clock synchronization on all machines in the system to provide a global system time
- directory service: provides record of the location of all resources in the system
- security service: provides authentication, authorization, and data protection
- distributed file service: provides a mechanism of accessing system files in a

transparent and consistent manner irregardless of the physical location of the file

- Remote Procedure Call (RPC): allows communication by having a client process make an RPC to a remote server process from which it desires a service. This RPC facility locates the server, binds to it, performs the call, and returns the server reply to the client in a manner which is invisible to the client.
- threads: allows the existence of multiple threads of control concurrently within a single process

These services and facilities are illustrated in Figure 2.3

2.4.2 CORBA

The development of OO technology has brought other issues into consideration as pertains to distributed computing. CORBA of the Object Management Group (OMG) is a response to the need to deal with the complexity of distributed objects in a system. Objects are discrete software components which contain data and can manipulate it. In CORBA, the services that an object provides are presented in a contract which acts as the interface between the object and the rest of the system. This contract:

- informs possible clients of the services which the object offers and posts how to construct a message to invoke the service
- notifies the communications infrastructure of the format of all messages the object will send and receive so that the infrastructure can provide transparent connection between sender and receiver.[8]

Each object possesses a unique handle which a client can send to the infrastructure to route a message to it. As objects move from one node to another in the distributed

Distributed Applications

Figure 2.3
Basic Services and Facilities Provided by DCE

system, they keep that handle. Thus each object in the distributed computing environment is identified by a unique handle and possesses a well-defined interface. A sending object passes a message to a receiving object by using the receiving object's unique handle and the message format is dealt with via the communications infrastructure. Figure 2.4 illustrates a client passing a message to a receiving object in the CORBA architecture. Note that both the client and the receiving CORBA object are isolated from the Object Request Broker (ORB) by the Interface Definition Language (IDL) interface, and that the message is managed by the ORB. This later fact ensures that distribution details are handled by the ORB, not the user [8,9]. A client may invoke any operation on any object that it can access over the network after it has located the object through a Naming Service (agent) within the ORB which knows each object's unique handle. The ORB receives the request targeting an object in the desired server and activates the server after checking its repository and determining that neither the server nor the object is currently active.

2.5 Performance Measurements

As greater demands are placed on computer systems to provide fast response, particularly in real-time systems, system performance becomes a critical issue. Designing systems with performance in mind is essential in order to meet the performance goals expected for the system. Some of the usual performance measures of interest are system throughput, response time, utilization, and service demands [10].

Obtaining these measures, however, is not always straightforward. Networks experience varying and competing loads resulting in large variation in system response times. To

Figure 2.4
Basic CORBA Communication Architecture

compensate for this effect, many data points need to be gathered to obtain a result which resides within an acceptable confidence interval. Resource consumption is often difficult to measure since operating systems may not provide a tool to extract the performance information of interest from the system. Other times, a tool may exist, but it does not provide information of sufficient granularity to be of significant value to the particular analysis. Performance analysis of a system needs to be carefully approached with these issues/limitations in mind.

The UNIX system contains a file system (`/proc`) which provides access to the image of each process in the system. Standard system call interfaces are used to access the files in this file system. The `ioctl()` system call allows the user to obtain information about a particular process and, depending on the options passed into this system call, information pertaining to system performance may be obtained. When the `PIOCUSAGE` option is specified, `ioctl()` returns process usage information such as CPU time consumed from the start of process execution. The `ioctl()` system call replaces the `Getrusage()` system call for Solaris 2.5.

A way to measure execution time for code is via a program profiler such as Quantify [32]. For code compiled under this tool's control, information from the compiler is used to obtain performance measurements at run-time. The compiler sections code into basic blocks and Quantify uses the compiler output to calculate its execution time based on the number of instructions it executes and the number of machine cycles used per instruction.

CHAPTER 3

THE ACE TOOLKIT

Frameworks reify the abstract concepts expressed in design patterns. The ACE framework implements a set of design patterns that ease the development of concurrent event-driven communication software. In the form of reusable C++ wrappers and frameworks, ACE simplifies the implementation of common communication software features such as event demultiplexing, event handler dispatching, connection establishment, and concurrency control [3][11]. This chapter provides a brief overview of the structure of the ACE toolkit and presents the design patterns implemented within ACE that were used in the design of the distributed system implemented for this thesis. In order to describe the design patterns in ACE, a consistent format as presented in [12], will be used to provide a common structure to the information. The template will set out a design pattern's name, intent, motivation, applicability, structure, participants, and collaborations.

3.1 Structure of the ACE Toolkit

The ACE toolkit, illustrated in Figure 3.1, possesses a layered structure that provides low-level C++ wrappers which encapsulate existing operating system (OS) mechanisms, and higher-level class categories and frameworks which implement common communication software design patterns. The C++ wrappers (see bottom of Figure 3.1) encapsulate and enhance the reuse of the following interprocess communication (IPC), OS concurrency, and virtual memory mechanisms [11]:

**Figure 3.1
Components of ACE Toolkit**

- IPC mechanisms
- event demultiplexing mechanisms
- multi-threading and synchronization mechanisms
- explicit dynamic linking mechanisms
- memory-mapped file mechanisms

By inheriting, aggregating, and instantiating the ACE C++ wrapper class libraries, applications use these components to increase their robustness by avoiding direct access to the C library interfaces [11].

The higher-level frameworks in ACE are shown in the upper portion of Figure 3.1. The ADAPTIVE Service eXecutive (ASX) framework uses the lower-level ACE C++ wrappers to provide:

- dynamic configuration and reconfiguration mechanisms
- communication mechanisms
- event demultiplexing mechanisms
- concurrency mechanisms

All these features are provided within a reusable OO architecture.

3.2 Design Patterns Implemented by ACE

This section presents the design patterns implemented within the ACE framework.

3.2.1 Reactor Design Pattern

3.2.1.1 Intent

The basic function of the Reactor pattern is to perform, within a single thread of control, concurrent event demultiplexing and event handler dispatching as events occur within event-driven applications. Application implementation is simplified in this pattern by separating application-independent code that demultiplexes events and dispatches pre-registered event handlers from application-specific code which uses user-defined methods in the event handlers [13].

3.2.1.2 Motivation

In distributed client-server applications, client requests for connection and service may arrive concurrently on multiple I/O handles to the server. The server listens on one I/O handle for the arrival of connection requests from new clients while maintaining separate I/O handles for each connected client. In order not to produce a detrimental effect on the response time for other clients, it is crucial that a single-threaded server not block for a lengthy period while reading information from a particular I/O handle. The Reactor pattern deals with this situation by accomplishing concurrent event demultiplexing and event handler dispatching within a single-threaded event loop as requests arrive at the server. The schism between event demultiplexing and event handler dispatching allows decoupling of application-independent and application-specific functionality. The application-independent components which demultiplex events and dispatch pre-registered event handlers can be reused thus allowing the user to develop only the application-specific methods in the event handlers [13].

3.2.1.3 Applicability

The Reactor pattern is of benefit when:

- events arrive concurrently from various sources and blocking or polling for event arrival on an individual source is wasteful;
- an event handler does not require blocking I/O when it exchanges messages with its peers;
- each message received by an event handler is processed within a short period of time;
- event demultiplexing using multi-threading is either infeasible, undesirable, or redundant
- the decoupling of application-dependent and application-independent components enhances functionality, portability, and extensibility.

3.2.1.4 Structure

The class diagram in Figure 3.2 outlines the structure of the Reactor pattern [13].

3.2.1.5 Participants

The classes/objects participating in the Reactor pattern include [13]:

- **Reactor:** An interface for registering, removing, and dispatching Event Handler objects is defined by this object. Mechanisms to support event demultiplexing and dispatching of application-specific event handlers must be provided in an implementation of this interface.
- **Event Handler:** This class specifies an interface used by the Reactor to dispatch callback methods defined by objects that are pre-registered to handle certain events.
- **Concrete Event Handler:** Implements the callback method(s) in an application-depen-

Figure 3.2
Class Diagram for Reactor Pattern

dent manner.

3.2.1.6 Collaborations

Events are associated with handles that are bound to event sources such as I/O ports, synchronization objects, or signals. When an event occurs, the Reactor activates Event Handler methods. In order to do this, the Reactor obtains the handle of the Event Handler subclass object (by invoking `Event_Handler::get_handle()`), registers this object, combines this handle with other registered Event Handlers, and waits for events to occur on the handle(s). Thus the respective Event Handler methods are located and dispatched upon event occurrence since the Reactor uses the handles as keys [13].

The message sequence diagram for the Reactor pattern as shown in Figure 3.3, illustrates how the collaboration using method callbacks is structured. The Reactor calls the `handle_event` method when an application-specific event occurs. As well, the `handle_close()` method is called by the Reactor to perform application-specific tidying which must occur before an Event Handler subclass object is removed [13].

3.2.2 Acceptor Pattern

3.2.2.1 Intent

Decouples passive initialization of a service from the tasks performed once a service is initialized.

3.2.2.2 Motivation

In systems where several types of data (status information, bulk data, commands, etc.) are routed between services executing on peers positioned in either LANs or WANs, interpro-

Figure 3.3
Message Sequence Chart for Reactor Pattern

cess communication must be coordinated. The Acceptor pattern accomplishes the following for network clients and servers that use connection-oriented communication protocols [14]:

- enables more flexible strategies for the concurrent execution of network services by allowing a service to run in a single- or multi-threaded fashion regardless of how the connection used to exchange data between peers was established;
- reduces the coupling between application-level service characteristics (such as application-level communication protocol and data format) and the strategies used to initialize the services;
- allows for portability of the connection establishment software across platforms that employ different network programming interfaces by parameterizing the code for accepting connections and executing services in such a way that these components may be extracted and replaced;
- protects passive-mode I/O handles from being used to read or write data by decoupling the connection-acceptance code from the code performing the service.

3.2.2.3 Applicability

The Acceptor pattern has application in the following situations [14]:

- The passive initialization of a service is not dependent upon the actual network service.
- Waiting for incoming connection requests on an individual peer by polling or blocking is wasteful in the case of concurrent connection requests arriving from various peers.

3.2.2.4 Structure and Participants

The classes/objects participating in the Acceptor pattern include [14]:

- **Reactor:** An interface, operating within a single thread of control, for demultiplexing connection requests received on one or more communication midpoints to the designated Acceptor.
- **Acceptor:** Creates and activates a `Svc_Handler` once it has passively accepted a connection from a peer. The Reactor automatically calls back to the Acceptor's `handle_input()` method, which implements the initialization of a `Svc_Handler`, when a connection request arrives for the Acceptor.

3.2.2.5 Collaborations

The three phases of the collaborations within the Acceptor pattern are shown in the message sequence chart in Figure 3.4 [14]:

- **Endpoint initialization phase:** A passive-mode endpoint bound to a network address is created and listens for connection requests from peers.
- **Service initialization phase:** The activation of a `Svc_Handler` occurs when a connection request is processed by the Reactor. The Reactor calls back to the Acceptor's `handle_input()` method which instantiates a new Concrete `Svc_Handler` object, accepts the connection into this object, and activates the object by calling its `open()` method which performs application-specific initialization.
- **Service processing phase:** The processing of application-specific requests that deal with the data exchanged between the `Svc_Handler` and its connected peer(s) occurs in this phase.

Figure 3.4
Message Sequence Chart for Acceptor Pattern

3.2.3 Connector Pattern

3.2.3.1 Intent

Decouples active service initialization from the tasks performed after the service is initialized [15].

3.2.3.2 Motivation

As discussed for the Acceptor pattern above, consider a system in which various types of data pass between services executing on peers located throughout LANs and WANs. A reliable, connection-oriented IPC protocol such as TCP is used to connect the peers to the agent (or mediator) coordinating interactions between the connected peers. The Connector pattern addresses the following issues in connection-oriented communication protocols [15]:

- Reuse of active connection establishment code for new services allows the independent evolution of service code from the code used to establish connections.
- Increase in portability of the connection establishment code across platforms is made possible by encapsulating lower-level network programming interfaces (such as sockets and TLI).
- Establishment of connections with a large number of peers can be accomplished efficiently by initiating and completing many connections in a non-blocking mode by using asynchrony.
- Flexibility of service concurrency policies is promoted since a service can execute in a single-threaded or multi-threaded manner once the connection is established regardless of the mechanism used to establish it.

3.2.3.4 Applicability

The Connector pattern has application in the following situations [15]:

- Active initialization of a service does not depend on the behaviour of the network service.
- Establishment of connections with a large number of peers connected over long-delay networks by an application is required.

3.2.3.5 Structure and Participants

There are three primary participants in the Connector pattern. Their structure is shown in Figure 3.5 and are described as follows [15]:

- **Connector:** Performs the connection and activation of a `Svc_Handler`. The active connection of the `Svc_Handler` with its remote peer is accomplished in the Connector's `connect()` method while the `complete()` method activates the connected `Svc_Handlers`.
- **Svc_Handler:** Provides a service with a generic interface. An endpoint (`peer_stream_`) for communication that encapsulates an I/O handle is contained within the `Svc_Handler` and is used to pass data between the `Svc_Handler` and its connected peer. This endpoint is activated by the Connector calling the `Svc_Handler`'s `open()` method once the connection is successfully established.
- **Reactor:** Allows the initiation and asynchronous completion of multiple `Svc_Handlers` by a Connector operating within a single thread of control.

3.2.3.6 Collaborations

The three phases of collaboration for asynchronous connections within the Connector pattern are shown in Figure 3.6 [15]:

Figure 3.5
Primary Participants in Connector Pattern

Figure 3.6
Message Sequence Chart for Connector Pattern
(Asynchronous Connections)

- **Connection initiation phase:** Connections, initiated either synchronously or asynchronously, are actively established for one or more `Svc_Handlers` with their peers. The mechanism for establishing these connections is determined by the Connector.
- **Service initialization phase:** After the connection associated with a `Svc_Handler` is established, the `open()` method of the `Svc_Handler` is called thus activating it. The differences between synchronous and asynchronous connection initiation as pertains to collaborations between components are shown in the message sequence charts in Figure 3.6 and Figure 3.7
- . Connection initiation and service initialization are shown within a single phase for the synchronous form while in the asynchronous form the two actions are within separate phases.
- **Service processing phase:** Application-specific activities which process the data passed between the `Svc_Handler` and its connected peer(s) occur after the above two phases are completed.

3.2.4 Active Object Pattern

3.2.4.1 Intent

Decouples method execution from method invocation so that synchronized access to a shared resource may be simplified when the method is invoked in different threads of control [16].

3.2.4.2 Motivation

In the Active Object pattern, a different thread is used for the execution of a method than was used to invoke it. This pattern provides the following advantages [16]:

Figure 3.7
Message Sequence Chart for Connector Pattern
(Synchronous Connections)

- Simplification of flow control: An active object can block in its own thread of control waiting for flow control to resolve still allowing other active objects to place messages onto a message queue associated with itself. This active object is then able to dequeue the next message from its message queue after it has processed its current request.
- Simplification of concurrent programming: Enqueue and dequeue operations on a message queue associated with active objects may occur concurrently. Serialized access to a shared resource is guaranteed due to the synchronization constraints.
- Parallelism is used: The inherent concurrency between active objects can be used to improve performance on multi-processor systems.

3.2.4.3 Applicability

The Active Object pattern has applicability in situations when [16]:

- Simplification of the design and implementation of a concurrent program is desired;
- Synchronized access to shared data by multiple threads of control is required;
- A difference in the order of method invocation from the order of method execution is acceptable;
- The operations performed on an Active Object are not so intricate that the synchronization, data movement, and context switching overhead is too high.

3.2.4.4 Structure and Participants

The structure of the major participants in the Active Object pattern is shown in Figure 3.8 and described as follows [16]:

- Client Interface: This participant is a Proxy which provides a method interface to client applications. When a method defined by the Client Interface is invoked, the cre-

Figure 3.8
Structure of Major Participants in Active Object Pattern

ation and queueing of a Method Object occurs.

- **Method Objects:** Whenever synchronized access to a shared resource maintained by the Scheduler is required by a method call, a Method Object is created. The context information within each Method Object allows the (1) execution of an operation after a method invocation and (2) the return through the Client Interface of results of the invocation.
- **Activation Queue:** Pending method invocations, represented as Method Objects, are maintained in a priority queue.
- **Scheduler:** The Activation Queue containing Method Objects desiring execution are managed by the Scheduler.
- **Resource Representation:** This resource object defines methods that are defined in the Client Interface and represents the shared resource being fashioned as an Active Object.
- **Result Handle:** A result handler is passed back to the caller when method invocation occurs on the Client Interface. This permits the method result value to be obtained after the completion of method execution.

After the Scheduler finishes the execution of a method invoked on the Client Interface, a Result Handle is returned to the caller allowing access to the method result value.

3.2.5 Lightweight OS Concurrency Mechanisms

Network servers on multi-processor platforms often experience heightened performance and simplified programming when a concurrent model of execution is employed. Using threads within such a server application to service multiple client requests avoids the fol-

lowing design alternatives [17]:

- Serializing client requests at a transport layer interface in an artificial manner;
- Internally queueing and handling client requests iteratively;
- Forking a new heavy-weight process (HWP) for each incoming client request.

When concurrency is a desired model of execution for a distributed application, ACE encapsulates the lightweight concurrency mechanisms provided by Solaris 2.x threads, POSIX Pthreads, and Win32 threads. The ACE OO thread encapsulation class library aims to meet the following requirements [17]:

- Avoid the “impedance” mismatch between OO and procedural programming when mixing C++ classes for concurrency and low-level C APIs provided as synchronization mechanisms with the operating system;
- Streamline application software design by allowing multiple tasks to progress independently using synchronous programming mechanisms;
- Use parallel processing potential of hardware platforms to heighten performance;
- Juxtapose computation with communication to improve performance;
- Decrease response time by assigning separate threads to different application tasks or services in interactive applications.

Thus the ACE OO thread encapsulation class library seeks to provide consistency of programming style and improve the portability and reusability of the base concurrency mechanisms without degrading performance.

The components of the ACE OO thread encapsulation class library in the ACE Locks Class category are briefly discussed as follows [17]:

- `Mutex`, `Thread_Mutex`, and `Process_Mutex`: classes which give simple mechanism to serialize access to a shared resource
- `RW_Mutex`, `RW_Thread_Mutex`, `RW_Process_Mutex`: classes which serialize access to shared resources which are changed much less than they are searched
- `Semaphore`, `Thread_Semaphore`, `Process_Semaphore`: classes which implement Dijkstra's "counting semaphore"
- `Null_Mutex`: locking interface used by other C++ wrappers for synchronization with no overhead
- `Token`: general-purpose synchronization mechanism which possesses `Mutex` properties as well as recursive mutex semantics.
- `Recursive_Thread_Mutex`: allows the nesting of acquire methods as long as the thread that owns the lock is the one re-acquiring it.

CHAPTER 4

PARODY OBJECT DATABASE

The growth of OO technology in this decade has seen it become the chosen paradigm for software development of many applications, with C++ becoming the language of choice for their implementation. While C++ incorporates many of the concepts of the OO programming paradigm, it does not support the existence of persistent objects. The ability to store and retrieve instances of classes (objects) automatically onto and from a persistent storage device has never been included as an integral part of the language. C++ programmers have needed to implement persistent objects themselves or use a database management system to do so.

The PARODY (Persistent, Almost Relational Object Database) ODMS, implemented as a C++ class library, establishes a persistent object database with a straightforward interface to its classes. The PARODY ODMS is founded on the relational data model. Building a database schema in PARODY is accomplished by defining a set of related application-specific classes derived from base classes in PARODY which provide persistent behavior. These application-specific classes encapsulate a description of their data members, specify key data members, and integrate the class with the PARODY ODMS. When supplied with the name of a database, PARODY creates and names two files one of which stores object data contents (with file extension “.DAT”), and the other which stores the index (with file extension “.NDX”) to the objects in the .DAT file. When supplied with the key data value

of the desired persistent object, these files are accessed and the PARODY ODMS performs the requested operation of the persistent object. The PARODY ODMS is designed for a single, interactive user on a single node [4].

4.1 PARODY Database Definition/Structure

A database is defined by designing classes that:

- possess application-specific data members and methods
- derive from the Persistent and Key base classes within the PARODY C++ library
- implement member functions that PARODY requires to maintain an object's persistence

The values of the key data members provide a unique identifier for a persistent object of a particular type. By invoking PARODY member functions at appropriate points and by implementing member functions that PARODY needs to carry out tasks requiring knowledge of the content and format of the persistent objects, application objects which inherit from the PARODY Persistent class assist in their persistence.

4.1.1 Persistent Object Class Definition

A persistent object in the PARODY database is the instantiation of an application-specific class derived from the Persistent abstract base class defined by PARODY. Such an object possesses data members and methods specific to the application class it represents yet it also possesses attributes by virtue of its descendance from the PARODY Persistent abstract base class. At a minimum, the derived Persistent class, from which the persistent object is instantiated, must have a primary key data member, a constructor, a destructor,

and two private member functions named `Read()` and `Write()`. Thus the only obvious distinction between a non-persistent class and a persistent class is the existence of the `Read()` and `Write()` member functions.

The constructor and destructor of the persistent object play an important role in its persistence. The constructor performs the following significant roles:

- initializes the base Persistent constructor with a reference to the Parody database object;
- initializes the persistent object's primary key data member to identify the object;
- calls the `LoadObject()` member function in the Persistent base class which locates and reads the object with the above specified key value from the PARODY database.

This last operation of the constructor lies at the heart of the PARODY scheme for retrieving objects, specified by their key value, from a persistent storage device. The `LoadObject()` function locates and reads an object from the PARODY database by calling the derived class's `Read()` function. This `Read()` function is defined as a virtual function in the Persistent base class thus all derived classes must provide an application-specific implementation of it. Data members are read into the persistent object as the `Read()` function calls the base class's user-provided `ReadObject()` function for each data member of the object. Thus the derived object and base Persistent classes alternate back and forth in this series of function calls as illustrated in Figure 4.1(a). The base Persistent class functions allow access into the persistent store in the .NDX and .DAT files using the application-specific information (data member size and type) provided by the derived class functions. Figure 4.2 is a message sequence chart which shows the invocations made by the con-

base Persistent class

derived object class

Figure 4.1 (a)

Flow of Method Calls to Input Persistent Object

base Persistent class

derived object class

Figure 4.1. (b)

Flow of Method Calls to Save Persistent Object

Figure 4.2
Message Sequence Chart for Constructor of Persistent Object in PARODY
ODMS

structor of an object derived from the base Persistent class[4].

The derived class destructor has the sole responsibility of calling the SaveObject() member function in the base Persistent class before completing its own destruction. An analogous flow of function calls occurs in the object destruction process as occurs in the object construction process. The base Persistent class's SaveObject() function saves new or changed objects in the PARODY database. It accomplishes this by calling the derived class's Write() function which is defined as a virtual function in the base Persistent class thus all derived classes must provide an application-specific implementation of it. Data members from the persistent object are written to the PARODY database as the base Persistent class's WriteObject() function is called for each data member by the Write() function in the derived class. Thus, as with the retrieval operation, the derived class's Write() function provides application-specific information of the object data members which is used by the base class functions to store the object contents on the persistent storage device. The sequence of function calls between the base Persistent and derived object classes is shown in Figure 4.1(b) [4].

4.1.2 Keys

In order to uniquely identify a persistent object in a PARODY database, each derived object class must have a data member designated as the primary key. The value of the primary key must be unique for each instance of an object of that class. Key data members descend from the PARODY base Key class thus giving them the behaviour of a key. The derived class constructor registers the key data member as a key during derived object construction. The base Key class defines various private member functions as pure virtual

functions. These functions, which must be user-implemented in a derived class, are needed by the PARODY indexing algorithm, the B-tree algorithm, to order and locate keys.

PARODY supports the existence of secondary keys and concatenated keys. While a persistent class may have only one object with a particular primary key value, numerous objects may possess the same secondary key value. A concatenated key contains two other key objects as its data values. When the relational equivalent of a connector file is required in the database design, the Persistent class will include a concatenated key as its primary key. Each of the key objects contained within the concatenated key is a primary key to another class and behaves as a secondary key to the connector class. Objects possessing a concatenated key may be retrieved on individual key values or on the combination of the two [4].

4.1.3 Class Relationships

When objects of one derived class are in implicit relationship with objects of another derived class, ie. a class has a secondary key of the same derived Key type as the primary key of another class, PARODY can be instructed to enforce that relationship. The degree of enforcement is dependent upon the application itself. PARODY is instructed of a relationship between objects of different derived classes through the invocation of the base Persistent class's Relate() member function from within the constructor of the related object. For instance, PARODY may prevent a persistent object from being added, deleted, or changed based on the nature of the relationship between that object and objects of another class. Integrity checks are incorporated into the PARODY database in this man-

ner.

4.1.4 Multiple-Copy Objects

Making in-memory copies of persistent objects can be fraught with problems since PARODY would not be aware of which copy to save on the database when the object goes out of scope. To avoid this situation, PARODY ensures that there is only one copy of any particular persistent object in memory at a time by using the Persistent class's handle/copy idiom [4] to implement reference counting. In this technique, a handle class is used as a class that an application instantiates. The handle class possesses a pointer to a copy class which holds all the members of the object as well as a count of the current number of copies. The constructor uses the new operator to create a new copy object when the first copy of the class is constructed. The address of the copy object is placed in the pointer of each new handle object and the counter in the copy object is incremented. Likewise, this counter is decremented when a handle class's destructor is invoked. The handle class deletes the copy object when the counter is zero. PARODY also substitutes the master copy of an object when it determines that the application is instantiating a copy.

4.1.5 PARODY File Architecture

A PARODY database consists of the:

- data file (file extension .DAT), which contains the objects
- index file (file extension .NDX), which contains key indexes to the objects.

These two files are created and opened when the application declares the Parody object.

The architecture of these files supports:

- variable-length persistent objects;
- indexes into and objects of multiple classes;
- persistent objects that may vary in length or may be deleted between the time an application declares one and when the application destroys it;
- reuse of deleted file space.

PARODY uses a system of file nodes to satisfy these objectives.

4.1.5.1 Nodes

An input/output system of fixed length file nodes in a thread manages both the data file and the index file in a PARODY database. Figure 4.3 illustrates the structure of a file showing the header record followed by the node records. The index files are comprised of fixed-length nodes while the objects in the data file are unformatted, variable-length data streams that may surpass the boundaries of a single node. They copy their data values into and out of the nodes.

The header record contains a

- pointer to the first node in a chain of deleted nodes,
- pointer to the highest node that has been allocated.

Each node record, a fixed-length disk record, possesses a node number followed by anything PARODY places into the node. The node number points to the next node in a logical node thread. Thus each node thread is composed of a number of nodes and PARODY maintains a pointer to the first node in the thread. Deleting and allocating nodes is performed by PARODY using the information contained in the header record [4].

4.1.5.2 Object File

Objects are stored in node threads under the PARODY scheme. An object begins on a node boundary and may occupy one or more nodes. The first node number of an object is the object's address. This file alone does not contain sufficient information to retrieve objects successfully. Class definitions and the member functions that support the objects provide the database with the types of the objects and their formats.

4.1.5.3 Index File

The address of a persistent object's first node is needed for PARODY to locate correctly any object. Object node addresses in the object file are associated with key values using indexes contained in the PARODY index file. One logical index table for every primary and secondary key in the derived Persistent classes is contained in the index file.

A table of index header records is used to represent each derived Persistent class. Each node in the index file is the header table for one of the derived Persistent classes in the database and thus is the first of a thread of index header nodes. In turn, each class header record is a table of index header records so that there is an index header record for each key in the class. The node number of the root node for the index and the length in bytes of the index key value are the data items contained in each index header record.

The B-tree data structure is used by PARODY to implement the indexes. The object node that corresponds to a specified key argument is located via this balanced tree of key values. Each node of the B-tree contains from one to a fixed number of keys in this hierarchy of nodes.

CHAPTER 5

SYSTEM DESIGN

Reuse of software through design patterns to obtain solutions to a particular software problem/issue is a stated objective of this thesis. The previous two chapters have introduced the major components which were used in the design of the **Extended Object Database Management System (EODMS)** developed in this thesis. This chapter

- states the mandate, assumptions, and issues of the EODMS design
- illustrates how design patterns were used to understand the functionality of these various components
- shows how design patterns were used to understand how component reuse and connection could best be accomplished
- presents the type of application-specific information which the user must provide to the EODMS

5.1 System Description

5.1.1 System Mandate

The EODMS designed for this thesis is comprised of a server part and a client part. The server is an object database server which receives requests from multiple, autonomous users to add, retrieve, delete, or change persistent objects. The EODMS server resides on a single node, however, the various users may be situated on any node on the network. The

client of the EODMS provides a user interface as well as coordinates the IPC between the user and the remote server.

5.1.2 Assumptions

The ODMS incorporated into the EODMS was the PARODY system described in chapter 4 of this thesis. The selection of this ODMS was based on PARODY's relatively small size (~2500 lines of code) as compared to other database systems, straightforward user interface, and existing documentation, as well as the fact that PARODY is written in and developed to interface with applications written in C++. This last factor allowed an easy port of the PARODY ODMS to the Solaris O/S and eased the liaison of this code with other software components written in C++ and already compiled and running under the Solaris O/S.

A major assumption from the onset of work was to leave the PARODY ODMS code as unaltered as possible in the development of the overall system. It was deemed consistent with the spirit of reuse to leave PARODY's simple user interface intact. In the case study presented in Chapter 6, it was assumed that a small company had been using the PARODY ODMS as it was originally intended-- interactively by a single user on a single node. The study in reuse develops when the company desires to expand its original database management system to handle multiple users distributed about numerous nodes on a network. The decision to leave PARODY's user interface virtually unaltered ensures that development of the EODMS will not reach within the database system and potentially and inadvertently corrupt valuable company records.

5.1.3 Issues

Three major issues, given the above assumptions, which arise in the development of the EODMS are:

- architecture of the EODMS;
- communication management as users of the EODMS may now reside on nodes different from the EODMS server;
- concurrency management as multiple users may submit requests to the server simultaneously.

5.1.3.1 System Architecture

The PARODY ODMS was originally designed to interface with a single user accessing the database from the same node on which the database resided. In the EODMS, this will no longer be the case. The EODMS will need to accommodate multiple users in a distributed user environment. It was deemed that a client/server system architecture would best characterize the action of the database system (server) in responding to requests from the various users (clients) [18, 19, 20, 21, 22].

Since the user was highly involved in the invocation of PARODY methods in the original ODMS, the choice of a client/server architecture for the EODMS necessitated the division of the EODMS into a “server” part and a “client” part. The architecture of the client/server “split” which was eventually chosen is discussed later, however, it is important to note that several possibilities existed although not all were desirable from a performance viewpoint.

5.1.3.2 Communication Management

In the EODMS, clients resident on different system nodes will send their requests to the server which may reside on yet a different node. In order for requests and replies to be sent between clients and server in this distributed system a communication mechanism must exist. The communication mechanism chosen for the EODMS is the ACE framework discussed in Chapter 3 [23]. The ACE Connector pattern, which actively initiates connections, is associated with the client, while the ACE Acceptor pattern, which passively accepts connections, is incorporated into the server.

5.1.3.3 Concurrency Management

The existence of multiple clients which may simultaneously request service from the database server necessitates concurrency control within the server. Lack of such a control mechanism could result in the corruption of database records due to two or more clients acting upon the same database record at the same time. In the EODMS, the database becomes a shared resource which must be protected by a mechanism which ensures serialized client access to the database records.

5.2 System Architecture

The general activities of the EODMS client and server are shown in Figure 5.1. The client is responsible for establishing a connection with the remote server, assembling and sending a request to retrieve, add, delete, or change a persistent object, and then receiving the response from the server. The server accepts connections from one or more simultaneously-connected clients, submits client requests to the PARODY ODMS in a manner

which incorporates concurrency control, and then returns the reply to the appropriate client. Design patterns were used to segregate the various duties which were required in both client and server and then identify a workable design solution reusing software components, PARODY ODMS and ACE, which were already available.

5.2.2 Client Architecture

The client architecture of the ODMS is shown in Figure 5.2 from a design pattern perspective. The three basic components in the client are the:

- Front-end transaction code (application-specific) which assembles requests to the database server;
- Proxy which encapsulates and isolates the network communication mechanism from the user;
- Connector which actively establishes connections with the EODMS server part.

Figure 5.3 provides a class diagram of the EODMS client while Figure 5.4 is an object diagram indicating the objects which exist at run-time in the client. A message sequence chart for the client is shown in Figure 5.5. .

5.2.2.1 Transaction Management Code (TMC)

This part of the client is the arena of the user. The user constructs application-specific objects, executes member methods, and works with the data members of these objects in this part of the client. All object operations which do not involve the server occur here. When it is desired to save the contents (data members) of an application-specific object, thus making the object persistent, the user invokes the `Add_Object()` method of the inter-

Figure 5.1
General Activities of EODMS Client and Server

Figure 5.2
EODMS Client Architecture from Design Pattern Perspective

Figure 5.3
Class Diagram for EODMS Client

Figure 5.4
Object Diagram for EODMS Client

Figure 5.5
Message Sequence Chart for EODMS Client (Retrieve Operation)

face to the EODMS. This action initiates further invocations which result in the object contents being written to persistent store. Similarly, if the user wishes to retrieve, delete, or change an object which resides in persistent storage, the user invokes the `Retrieve_Object()`, `Delete_Object()`, or `Change_Object()` methods, respectively, providing the primary key value of the persistent object on which the operation is to occur.

After the user invokes the appropriate EODMS method and the server performs the request, a success flag is returned to the transaction code notifying the user of the success or failure of the request. If a Retrieve operation was specified, a pointer to an instantiated object whose contents are those of the desired persistent object are returned to the user. If the user submits a primary key value which does not correspond to the primary key value of a persistent object when requesting a retrieve, the user is notified by receiving a NULL pointer instead of a pointer to an instantiated object.

5.2.2.2 Proxy Pattern

It would be tedious, error-prone, and not conducive to reuse to require the user to implement and/or interface directly with the interprocess communication protocols needed to connect a client with a remote server. Access to a remote server should be transparent and straight-forward for a client. It is preferable that a client not be required to significantly change its calling behavior and syntax from that used to invoke a local server when invoking a remote version of a server. A design pattern which addresses these issues is the Proxy.

The Proxy design pattern [24] is comprised of three participants: the client, the proxy, and the original. The original implements a service whose functionality the client wishes

to invoke. However, there may be reasons why the client may not directly be able to invoke this functionality ie. the server may be remote from the client. The proxy acts as a bridge between client and server by providing the client with the same interface as the original thus allowing the client to preserve the calling behavior and syntax needed by the original.

Figure 5.6 is a message sequence chart for the Proxy pattern showing the following typical dynamic scenario [24]:

- client submits a request for service to the proxy;
- proxy receives this request and performs any preprocessing required before forwarding the request to the original;
- proxy sends the request to the original using the proper communication protocols and security measures;
- original receives the request, fulfills it, and sends the reply to the proxy;
- proxy receives the reply from the original and performs any appropriate post-processing before forwarding the reply to the client.

In the design of the EODMS there was a requirement for a client(s) to submit requests to a remote database server. A remote proxy, which possessed the same interface as the original database server, was employed in the client to encapsulate and maintain the physical location of the original thus hiding the IPC mechanisms from the user. The proxy and original are associated by giving the proxy a handle to the original (ie. a socket provided by the Connector pattern). By decoupling clients from the location of the server, clients are not adversely affected by changes in the location of the server or in the networking

Figure 5.6
Message Sequence Chart for Proxy Pattern

infrastructure. Thus stability and reusability of the client code is enhanced [24].

5.2.2.3 Connector Pattern

The Connector Pattern as implemented in the ACE framework has already been presented in Chapter 3. This pattern was implemented in the EODMS client to actively establish IPC connection between a client and the remote server. The handle for the EODMS server obtained by the ACE connector was provided to the proxy to allow the transparent communication of the client with the server.

5.2.3 Server Architecture

The design patterns used to formulate the server architecture are shown in Figure 5.7.

The major components in the server are the:

- Acceptor which passively accepts a client connection;
- Reactor which demultiplexes the requests arriving on the accepted client connections and dispatches the appropriate event handlers;
- Active Object which is used to implement a pool of threads which are responsible for handling client requests;
- ACE concurrency mechanism;
- original PARODY ODMS.

The collaborations among the participants in the various patterns are illustrated in the object diagram in Figure 5.8 and are explained in more detail in the following sections. A class diagram for the EODMS server is shown in Figure 5.9 while Figure 5.10 provides a message sequence chart for the server showing object interactions for the following four

Figure 5.7
EODMS Server Architecture from Design Pattern Perspective

Figure 5.8
Object Diagram for EODMS Server

Figure 5.9
Class Diagram for EODMS Server

Figure 5.10(a)
Message Sequence Chart for EODMS Server
Connection Initialization Phase

Figure 5.10(c)
Message Sequence Chart for EODMS Server
Service Processing Phase

phases of operation in the server

- connection initialization
- service initialization
- service processing
- service completion and server shutdown

5.2.3.1 Acceptor and Reactor

The interplay between the Acceptor and Reactor patterns has been described in Chapter 3 and the application of these patterns to the EODMS is as presented in that earlier chapter.

When the server part of the EODMS commences execution, its first principle activity is to establish and initialize a passive endpoint of communication and register that endpoint with the Reactor. This endpoint initialization phase involves the creation of a passive-mode endpoint bound to a network address which listens for connection requests from peers. When a connection request arrives, the Reactor issues a callback to the Acceptor which causes a concrete `Svc_Handler` object to be created. The connection is accepted into the new `Svc_Handler` object and the `Svc_Handler` object is activated and service-specific initialization occurs. The service processing phase performs application-specific tasks when the above phases are complete.

In the case of the EODMS, the service processing phase involves the submission of the client request to a thread pool which has been spawned within the server part. Multi-threading the server allows concurrent execution of client requests by assigning each incoming request to a separate thread of control. The server uses the Active Object pattern to implement the thread pool [20][25].

5.2.3.2 Active Object

The Active Object pattern is employed to simplify synchronized access to a shared resource by methods invoked in different threads of control. Since this pattern decouples method execution from method invocation, as discussed in Chapter 3, separately executing threads may access the shared resource in an interleaved manner [16].

The specific application of the Active Object pattern to the EODMS is shown in Figure 5.11 [25]. Client requests to retrieve, add, delete, or change a persistent object arrive at the server. The implementation of the Acceptor and Reactor patterns perform the connection and Svc_Handler creation activities on the server side. Once activated, the new concrete Svc_Handler enqueues the client request into a global request queue contained within the Active Object. As a client request comes to the head of this queue, it is dequeued and, if a thread is available in the thread pool, is given its own thread of control and allowed to proceed with a controlled database access. If all threads are active when a client request comes to the head of the request queue, the request is blocked until a thread is available. Once a client request has completed a controlled access to the database, the Active Object returns the results of the database access to the appropriate client. The thread for that client request is returned to the thread pool to be available for use by a client request on the global request queue.

In addition to the thread pool model, numerous models for multi-threaded servers exist[20]:

- *Thread per request* - The server spawns a thread for each incoming client request
- *Thread per client* - The server spawns a thread for each client which the client exclu-

Figure 5.11
EODMS Server Architecture

sively uses during the entire time it is connected with the server

- *Thread per object* - Every service in the server (associated with an object) is executed in its own thread of control.

The Thread Pool implementation was chosen over the above options because it is advantageous for scenarios in which OS resources are limited. By pre-spawning the pool of threads, resource consumption is minimized [25].

5.2.3.3 Concurrency Mechanism

Once a client request has been assigned a thread from the thread pool, reception of the data for the client request occurs via ACE and processing of the client commences. Multiple clients, each possessing their own thread of control, may be active in the server external to the PARODY section at this point. Each of these threads must access the PARODY interface methods in order to accomplish the task its associated client desires. The PARODY ODMS, originally designed for a single user, has no means of enforcing serialized access to itself, thus in a multi-threaded environment corruption of database records may occur as multiple operations may be simultaneously attempted on a single persistent object. To avoid this scenario, incorporation of a concurrency mechanism was necessary.

The ACE Toolkit provides various lightweight OS concurrency mechanisms. The category of interest, as described in Chapter 3, is the RW_Mutex provided by the ACE Locks Class category. Multiple threads may obtain a reader/writer lock simultaneously for reading, but only one thread at a time may acquire the lock for writing purposes. Concurrent execution is improved when the shared resources protected by the mutex are read more often than they are written. This situation accurately represents the scenario dealt with in

the case study presented in the next chapter. The database files, as shared resources, far more often experience retrieve operations than add, delete, or change operations in that study.

The grain of locking within the server implementation was not a straightforward design decision. Originally locking only those portions of PARODY ODMS which handled the database file accesses was considered, however, it quickly became apparent that the PARODY system, designed for a single user, possessed objects which would need to be global to all the threads yet whose methods were not accessed via a single interface throughout the code. In order to implement finer-grained locking (ie. locking groups of 100 nodes in the .NDX and .DAT files), locks within the PARODY ODMS would be frequent and fragmented. The performance overhead of locking many small pieces of code began to outweigh the benefit of increased concurrency within the server. Thus the final server design simply places a RW_Mutex around the entire PARODY ODMS within the EODMS. Because of the manner in which ACE implements the RW_Mutex, preference is given to writers. If multiple readers and a single writer are waiting on the lock, the writer will acquire it first.

5.3 User Involvement in the EODMS

In the original PARODY ODMS, the user was required to provide application-specific classes which defined the data members and key data members of a persistent object. The PARODY ODMS used this information to develop a schema of the database on which it was required to operate. Accessing key values by the B-tree indexing mechanism, reading and writing object data members to/from the persistent store, setting up relationships

between persistent objects for integrity purposes all required this application-specific information.

As well, within the application-specific classes, the user implemented methods to retrieve, add, delete, and change persistent objects by directly accessing the PARODY ODMS interface methods. This approach was deemed to be inefficient and error-prone. For each class of persistent object the user must implement a separate method to perform a specific operation. The only difference between methods to perform a specific operation on different classes of persistent objects was the application-specific data (ie. class name and data member names). In the EODMS, the functionality of the method performing a specific operation was captured and means of providing the application-specific data to it other than requiring the user to hard-code it into the method was achieved. The original approach was also prone to error since incorrect use of the PARODY interface methods by the user was possible.

Thus a major consideration in the design of the EODMS was how to reduce and consolidate the information required from the user. Since the interface to the original PARODY ODMS was maintained, the user was still required to provide a derived Key class with associated member methods for each class of persistent object as needed by the B-tree indexing mechanism. As well, the derived Persistent classes with their constructor, destructor, Read(), and Write() member methods were also still required to be implemented by the user since these methods are integral to the original PARODY ODMS and their removal would necessitate a re-design of the original PARODY ODMS. However, the EODMS has relieved the user of providing the methods to perform specific operations

(add, delete, retrieve, change) on application-specific persistent objects. These methods have been templated and, using the application-specific data provided by the user in a few header files and straight-forward application-specific methods in the derived Persistent classes, the EODMS hides the PARODY interface from the user almost entirely. This relieves the user from having to understand the logic of the PARODY ODMS, as was required in the original system, and thus greatly reduces the possibility of an implementation error. This approach also eliminates the wasteful repetition of the original PARODY ODMS in requiring the user to implement a separate, virtually identical, method to perform the same database operation for each derived Persistent class.

The Graphic User Interface (GUI) of the original ODMS has been removed. The EODMS is not interactive. The user interacts with the EODMS via the transaction management code in the client (user-supplied C++ code).

The following sections outline the information required by the EODMS client and server in the form of header files and application-specific class methods.

5.3.1 User Involvement in Client Part of EODMS

User involvement in the client comes in three forms:

- header files
- application-specific classes with member methods
- transaction management code

In examining the above three forms of user involvement it is important to realize that the client has no direct interaction with the PARODY ODMS and thus has no derived Persis-

tent class objects. Thus the user need not implement any of the virtual methods in the Persistent class within the client. The client creates objects of the desired application-specific classes and sends/receives these object contents and transaction information in an appropriate form to/from the server via ACE.

5.3.1.1 Header Files

The user is required to provide two header files (see Appendix 1):

- `application_client.h`
- `config.h`

The `application_client.h` header file

- defines application-specific macros needed to instantiate templates within the server;
- includes the implementation (`.cpp`) files which have been provided by the user and will be described in the next section.

The `config.h` header file defines application-specific structures, one for each class of persistent object, whose data members are the data members of that derived Persistent class. These structure definitions are used to encode/decode the persistent object contents which are transmitted as a bit stream over the network to/from client to/from server via ACE. ACE is not capable of sending or receiving an object per se. The client must extract an object's data members, pack them into an application-specific structure, and then send this structure via ACE to the server. Likewise, when the client receives from the server a reply containing persistent object contents, the client must pack an application-specific object with the data members received via ACE as an associated structure.

5.3.1.2 Application-Specific Classes

On the client side, the user must provide header (.h) and implementation (.cpp) files for each class of persistent object which define data members and member methods to set the values of these data members and return their values to the user. In addition, the user must implement the following two methods which allow the remote transaction to occur:

- *Any_struct* Pack_Send_Struct()*

This method creates a structure of type *Any_struct* with data members identical to the data members of its associated object (ie. object of type *Employee* has an associated structure type of *Employee_struct*). The structure data members are packed using the corresponding data members of the associated object. Use of this method occurs as a client is preparing to send a request to the server via ACE. This method is used regardless of which operation is specified.

- *void Pack_Object(AI_struct* ais)*

This method sets the data members of an object whose contents have been received from the server via ACE at the end of a transaction. The data members are extracted from the buffer used by ACE to transmit data from server to client. This method is only invoked during Retrieve operations.

5.3.1.3 Transaction Management Code (TMC)

The TMC is the *main()* function of the EODMS client. The initial TMC reads command-line arguments and connects the client to the remote server by specifying machine and port address of the server (received as command-line arguments) and the address of the *Request_Handler_Client* (derived class of *ACE_Svc_Handler*). Once this initialization is

complete, the TMC performs its major function which is to create and initialize application-specific objects and invoke Proxy object methods corresponding to the desired EODMS operation to be performed on these objects. Once the Proxy object returns the server reply, the client may proceed with another request. At the completion of all transactions for a client, the TMC deletes the Request_Handler_Client and closes the client connection to the server.

5.3.2 User Involvement in the Server Part of EODMS

User involvement in the EODMS server comes in two forms:

- header files
- application-specific derived Persistent classes with member functions

The server has direct interaction with the PARODY ODMS. Thus the persistent application-specific objects dealt with in the server part are derived Persistent class objects. In this section of the EODMS, the user must provide the virtual methods of the Persistent and Key classes as needed by PARODY.

5.3.2.1 Header Files

The user is required to provide two header files (see Appendix 2):

- config.h
- application.h

The config.h file is the same header file described in the section above for the client. The application.h file is similar to the application_client.h for the client, however, it contains more information needed by the server. This header file includes the application-specific

derived Persistent class header files and defines macros of application-specific information needed by templated functions within the server. It also includes the implementation of a small function, `Establish_relation()`, which is used by the PARODY ODMS during its initialization.

5.3.2.2 Application-Specific Derived Persistent Classes

The user is required to provide all the necessary application-specific derived Persistent and embedded derived Key classes with their associated member methods as for the original PARODY ODMS. The five following additional member methods in each derived Persistent class are needed in the EODMS:

- *Input_Object(`AI_struct *ais`)*

This method is invoked when the server desires to fill the data members of a newly instantiated derived Persistent object with contents sent via ACE from the client. Only an “add” transaction causes this method to be invoked.

- *Key_Value(`AI_struct* ais, int* key_value`)*

This method extracts the primary key value from the buffer contents sent from the server to the client via ACE. It is used whenever a new client request arrives.

- *Pack_Reply_Struct(`Any_struct *any_struct`)*

This method fills the application-specific structure which will be returned to the associated client via ACE with the contents of the desired object resident in the persistent store. Only a “retrieve” transaction causes the invocation of this method.

- *MakeChange(`AI_struct *ais`)*

Called in the case of a “change” transaction, this method extracts the revised data contents

sent by the client for a persistent object from the buffer and reassigns the data members of the persistent object with the revised data.

- *Constructor(int *key_value)*

This version of the constructor accepts as input an array of primary key values and uses the appropriate array members to initialize the desired key values for that specific persistent object. The Persistent class's LoadObject() method is invoked. This constructor was implemented to accommodate concatenated primary keys.

CHAPTER 6

PERFORMANCE MEASUREMENTS AND RESULTS

Understanding the performance implications of design decisions in OO systems becomes more important as OO technology continues to grow. Trade-offs considered during the design phase of a project are based on many factors and one factor of significance is the performance of the system. As systems become more complex and response time more critical, developers and users alike do not want to compromise system performance due to the benefits provided by an OO design and implementation.

This chapter presents a brief performance study of the EODMS designed and implemented in this thesis. Test suites and results showing system throughput under a varying load for different:

- numbers of threads in the EODMS server thread pool
- locking scenarios for the PARODY ODMS within the EOMDS server are presented

Server service time is provided for retrieve and change database operations. A classification of server CPU consumption into exponential, hypo-exponential, or hyper-exponential is made based on the results of this later study. This experimental classification of server service time was undertaken since many analytical models assume an exponential service

time [26, 27, 28].

The case study presented in this chapter is a personnel system for a small company of:

- 1000 employees
- 7 departments
- 4 projects
- 1000 assignments

The company desires to store information in a database pertaining to its employees, departments, projects, and assignments. Quantities to be maintained are the:

- name and employee number of each employee;
- department number of each employee;
- name and department number of each department;
- manager of each department;
- name and project number of each project;
- manager assigned to head each project;
- number of hours expended on each project;
- employee assigned to each assignment;
- project under which each assignment occurs;
- hours expended on each assignment.

6.1 Class Presentation and Set-up

The above information was apportioned into four application-specific classes:

- Employee

- Department
- Project
- Assignment

A brief discussion of the data members, key data members, and relationships between classes for each application-specific class is given in the following sections. Recalling that the application classes are derived from the Persistent base class in the EODMS server, the discussion will center on the class definitions provided on the server side. The client side definitions possess the same data members but without the derived Persistent and Key class methods required by the PARODY ODMS.

For each of the above classes, the definition and implementation in the EODMS client occurs in the *<class_name>_client.h* and *<class_name>_client.cpp* files. The definition and implementation in the EODMS server takes place in the *<class_name>.h* and *<class_name>.cpp* files.

6.1.1 Employee class

The Employee class uses the handle/copy idiom [4] to permit the instantiation of more than one copy of the same object. This class is derived from the Handle class and implements an interface to the EmployeeBody class via member methods. The EmployeeBody class, which specifies the Employee class as a friend class, is derived from the Persistent base class and contains application-specific data members and methods. The methods implemented in the EmployeeBody class are those required by the PARODY ODMS (see Chapter 4) while the data members are:

- employee number

- department number of employee
- employee name

The data member specified as the key is the employee number. Since this number is different for each employee, it provides a unique index by which the PARODY ODMS can reference each Employee object. As the primary key, this data member must be derived from the PARODY Key class.

As well, the EmployeeBody class states that there is a relationship between itself and the Department class. The statement of this relationship prevents the addition or change of an Employee object where the department number is non-null and represents a Department object which does not reside in the database.

6.1.2 Department class

The Department class has a straightforward definition and data members in this class are:

- department number
- department name
- employee number of the department manager

The primary key for this class is the department number since this is unique for each department. The constructor of the Department class specifies that this class is related to the Employee class. This ensures that no addition or change to a Department object may occur if the employee number of the department manager is non-null and represents an Employee object not resident within the database.

6.1.3 Project class

The data members in the Project class are:

- project number
- project name
- employee number of project manager
- hours-to-date expended on project

Since each project is assigned a unique project number, this number is selected as the primary key data member for the class. Again, the Project class is related to the Employee class to ensure that no addition or change to a Project object will take place if the employee number of the project manager is non-null and represents an Employee object not in the database.

6.1.4 Assignment class

The Assignment class provides an example of a many-to-many relationship in that it is a relational connector class between Employee class objects and Project class objects. This class has data members:

- assignment number
- hours-to-date expended on assignment

The assignment number data member is a concatenated key derived from the base Key class and contains two member objects, employee number and project number, which are themselves primary keys for two other classes. Thus, to specify a unique value for the assignment number key data member, unique values for the employee number key data

member and the project number key data member must be provided.

The assignment class is related to the Employee class and the Project class.

6.2 Collection of Performance Measures

System throughput and server service time characterization [29] were obtained in the course of this thesis. The subsequent sections present the methodology and the test suites used to obtain these data. The script used to process the performance measurements is presented in Appendix C. This script controlled the:

- removal of old logging directories and the creation of new ones,
- editing of the client(s) and server command lines via the UNIX gawk facility,
- launching of the client(s) and server processes,
- post-processing of data via Matlab mathematical software.

6.2.1 System Throughput

The following scenario was executed for different numbers of clients (1 - 24) for the following cases:

- write_lock on both retrieve and change operations (1 thread in server thread pool).
Two cases were considered: (1) 100% retrieve operations, and (2) 100% change operations.
- read_lock on retrieve operation and write_lock on change operation (1,4,8 threads in server thread pool) for a retrieve:change operation ratio of 75:25.

The clients used in this case study were independent and homogeneous. Each client pro-

gressed through a cycle which included an exponential think time (mean of 100 msec) plus a request for a single database operation (transaction). The first scenario above provided bounds on system throughput for various ratios of retrieve:change operations. In the second scenario the client was restricted to retrieve or change operations in a ratio of 75 (retrieve):25 (change). This ratio was arbitrarily chosen as representative of the database operations of the small company in the case study. Retrieve operations would be more prevalent than change operations and add/delete operations would account for a very small portion of total database operations. Add/delete operations were also excluded as it was desired to leave the size of the database unaltered in order that access times be consistent from one client cycle to the next. Clients randomly chose which operation to request according to the 75:25 ratio. The application-specific object (Employee, Department, Project, Assignment) on which this operation occurred was randomly selected with uniform probability.

Each client repeated this cycle for at least 3000 times (100 transactions per test; 30 tests) in order to compensate for the variation in client throughput due to network traffic. Client executions were conducted at periods when transience in network traffic was deemed to be lowest. Client and server processes executed on the Real-Time and Distributed Systems (RADS) Quiet Network in order to reduce the effect on performance data of transient loads on the host machines.

The UNIX function *gettimeofday()* was used to record client cycle time from which client throughput could be obtained.

6.2.2 Server Service Time Characterization

A characterization of server service time for retrieve and change operations was obtained. The amount of CPU time required for these operations was collected over 3000 server accesses per operation. Each test for an operation consisted of 100 transactions which were repeated for 30 times to account for variations due to transient load on the network. The mean and coefficient of variation of the average CPU time consumed for that operation were calculated. The coefficient of variation of a random variable X is denoted by C_x and defined by:

$$C_x = \frac{(\sigma_x)}{E[x]}$$

The coefficient of variation of an exponential random variable is 1, so C_x is a measure of deviation from the exponential distribution [27]. This data was gathered using one thread in the EODMS server thread pool and by calling the UNIX *ioctl()* system call to obtain CPU time consumed during the database operation.

6.3 Results and Analysis

6.3.1 System Throughput for Write_Lock Case

Results of experiments conducted to determine system throughput where retrieve and change database operations were protected by a write_lock is shown in Figure 6.1. The two curves generated show system throughput for the cases of one thread in the server thread pool and (1) 100% retrieve operations, and (2) 100% change operations. These

Figure 6.1
System Throughput vs. Number of Clients
(Write Lock on Retrieve and Change Operations)

curves provide bounds on system throughput for any retrieve:change operation ratio selected. As seen, system throughput rises in characteristic form to maximum values of approximately 36 responses/sec (change operation) and 54 responses/sec (retrieve operation). The size of the confidence interval at the 95% level is approximately $\pm 1\%$ about the mean. This case is presented as a basis from which to compare the read/write_lock case in the next section.

6.3.2 System Throughput for Read/Write_Lock Case

Results of experiments conducted to determine system throughput where retrieve operations were protected by a read_lock and the change operations by a write_lock are shown in Figure 6.2. All curves rise in characteristic fashion with maximum throughput values of approximately 42 (1 thread), 46 (4 threads), and 49 (8 threads) responses/second. The size of the confidence intervals at the 95% level is $\pm 1\%$ to $\pm 5\%$ about the mean except at 12 clients (4 and 8 threads) where the size of the confidence interval at the 95% level is $\pm 10\%$ about the mean. These mid-range points displayed more instability than points at lower and higher client numbers. Figure 6.3 shows the set of response time curves corresponding with this scenario. Response time is defined here as client cycle time less client think time.

The benefit of multi-threading is seen. As the number of threads in the EODMS server thread pool increases from one to 8, there is an increase in system throughput of approximately 9% (1-4 threads) and 6% (4-8 threads). This increase, although present, is not substantial and indicates that the database operations are heavily CPU bound with little disk I/O occurring. It is concluded that because the database size is relatively small, it has been

Figure 6.2
Throughput vs. Number of Clients
(Read_Lock on Retrieve/Write_Lock on Change)

Figure 6.3
Response Time vs. Number of Clients
(Read_Lock on Retrieve/Write Lock on Change)

largely or entirely read into cache. Database file accesses occur more quickly when the file is stored in cache than if a disk I/O operation occurred to access data. Thus the benefit to throughput that multi-threading would have brought if significant disk I/O occurred has been diminished.

A slight decrease in system throughput is seen from 12 to 24 clients for the 4 and 8 thread cases. This is attributed to an increase of collisions on the network due to the increase in the number of clients.

When the results for the cases in which there is one thread in the thread pool for both the write_lock case and the read/write_lock case are compared, it is observed that the system throughput for the read/write_lock case falls within the range established in the write_lock case.

6.3.3 Service Time Characterization of EODMS Server

Table 6.1 shows the results obtained for server service time characterization for retrieve and change operations. The average CPU time consumed during a retrieve and a change operation is 6.2 msec and 13.5 msec, respectively. The coefficient of variation for both these values is less than one. This indicates that the server service time for these operations is hypo-exponential ($C_x < 1$) [27]. The distributions of CPU service times for retrieve and change operations are shown in Figure 6.4 and Figure 6.5, respectively. For 2400 transactions submitted to the server, retrieve operations require approximately 4 - 12 msec to process with the majority needing 6 or 9 msec to perform. Change operations require approximately 10 - 40 msec to perform with the majority requiring 10 or 20 msec

Retrieve Operation		Change Operation	
Average CPU time (msec)	Coefficient of Variation	Average CPU time (msec)	Coefficient of Variation
6.2238	0.31562	13.527	0.46102

Table 6.1
Service Time Characterization of EODMS Server for
Retrieve and Change Operations

Figure 6.4
CPU Service Time Characterization for EODMS Server
(Retrieve Operation)

CPU Service Time (microsec) in EODMS Server (Change Operation)

Figure 6.5

**CPU Service Time Characterization for EODMS Server
(Change Operation)**

require approximately 10 - 40 msec to perform with the majority requiring 10 or 20 msec to process.

CHAPTER 7

CONCLUSIONS

Reuse of successful solutions to known software problem appears to be a desirable and efficient track on which to proceed when developing new software systems, however this path has pitfalls and obstacles which must be understood and appreciated or the unwary designer may find himself/herself facing difficulties which had not been anticipated. This research attempts to explore some of the practical challenges encountered when software reuse is attempted. Using experience gathered during a reuse case study, the following sections highlight issues a designer may need to overcome when reusing software. Conclusions of a brief performance analysis on the case study end this chapter.

7.1 Difficulties in Software Reuse

The following issues were encountered during the software reuse case study which lessened the ease with which reuse could occur.

7.1.1 Impedance Mismatch

When the connection of two or more software components is attempted in order to produce a new component, incompatibilities in their respective interfaces may thwart a simple connection. To accomplish the marriage of such components, interstitial code which

adapts the interface of one component to that of the other component(s) is required in order that communication and connection may take place. This adaptation exercise may require the designer to delve into the code of the individual components more than desired which defeats a primary goal of software reuse which is to have software components be as “plug in” as possible. Increased designer understanding of the individual components at the code level consumes valuable time in the current product's development cycle which may best be used elsewhere.

7.1.2 Differing Internal Assumptions in Components

Another frustrating difficulty which emerges when attempting the connection of existing software components is that the individual components have been designed with certain architectural and functional assumptions. A designer may possess an existing software component which has much of the functionality desired for a component in a system under development. However, if the existing component was designed for a single node only and the new system must support distribution, or if the new system must support concurrency and the existing component was not designed with this in mind, the designer is left with having to incorporate the additional functionality into the existing component. Again this requires a more in-depth knowledge and understanding of the existing component on a code-level. Resources are consumed as the existing component is re-designed in order to accommodate the required functionality. Such re-design is not always straightforward and numerous attempts may be needed before an adequate and efficient design is obtained.

7.1.3 Differing Internal Assumptions of Control

A subtle issue which may arise when reusing software is that of control. Individual com-

ponents will have been designed with some assumption of the level of control that they possess. When integrating numerous such components into a new design, there may be "power struggles" between such components. Disentangling this conflict and establishing an orderly hierarchy of power can present a considerable challenge to the designer.

7.2 Overcoming Difficulties in Software Reuse

The following sections discuss the benefits of OO technology and Design Patterns in relation to overcoming the difficulties in software reuse.

7.2.1 Benefits of OO Technology

The object-oriented (OO) paradigm is to improve the reliability, maintainability, and reusability of code through the concepts of information hiding, inheritance, and polymorphism. The well-defined interface which an object presents via its public methods provides improved visualization and understanding of the interaction between objects. As well, if software is designed with appropriate hinges [30], specific, designated places where the software is designed to adapt, future modifications with one of these hinges is more easily accommodated. The combination of increased understanding of the software system via better visualization of object interaction, as well as the greater adaptability permitted by hinges, a system designed under the OO paradigm supports greater possibility of reuse.

7.2.2 Benefits of Design Patterns

Design patterns are abstractions which describe successful solutions to common recurring software problems [31]. Their purpose is to communicate architectural knowledge, often

resident only in source code or in the heads of expert designers, to the public arena so that reuse of the proven design is possible. Patterns are not infinitely flexible and thus have a context in which they apply. A designer states this context as well as the benefits and costs of using the pattern. This allows a clearer decision as to when or when not to use a particular pattern.

7.3 Performance Measurements

Average system throughput and response time were the performance measures obtained for the EODMS designed for the case study presented in this thesis. These measures were gathered for systems of 1-24 clients with 1, 4 and 8 threads in the EODMS server thread pool using retrieve and change operations in a ratio of 75 (retrieve):25 (change). System throughput for this retrieve:change ratio in the one thread case fell within the measured experimental bounds. The benefits of multi-threading the server were present but somewhat minimal. It is concluded that this is due to the small size of the database. The database files are small enough to be largely present in cache thus minimizing the amount of concurrency achievable if more I/O operations were required to access data.

Average server service time for retrieve and change operations, respectively, was measured. For both operations it was determined that the service time is hypoexponential in nature.

7.4 Future Work

One of the goals of this work was to establish a distributed client/server system (EODMS) with a non-synthetic server on which performance measurements could be obtained. Hav-

ing achieved this goal, the following future work can be conducted:

- conduct more in-depth performance study of the EODMS
- build a Layered Queuing Network (LQN) model of the EODMS
- analyze performance characteristics of the LQN model and compare results to those obtained from the actual system both in the preliminary study conducted in this work and the more in-depth study recommended

In this manner a more complete understanding of the performance characteristics of OO systems designed on the basis of Design Patterns can be explored and developed.

REFERENCES

- [1] Stephen Prata,, "C++ Primer Plus", Waite Group Press, Corte Madera, CA, 1995.
- [2] D.C. Schmidt, Lecture Handout, Carleton University, 1997.
- [3] D.C. Schimdt, "ACE: An Object-Oriented Framework for Developing Distributed Applications", Proceedings of the 6th USENIX C++ Technical Conference, Cambridge, MA), USENIX Association, April 1994.
- [4] Al Stevens, "C++ Database Development", MIS: Press, New York, New York, 1992.
- [5] Mary E.S. Loomis, "Object Databases, The Essentials", Addison-Wesley Publishing Company, Don Mills, Ontario, 1995.
- [6] Catherine Ricardo, "Database Systems: Principles, Design, and Implementation", Macmillan Publishing Company, New York, New York, 1990.
- [7] Andrew S. Tanenbaum, "Distributed Operating Systems", Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1995.
- [8] Jon Siegel, "CORBA Fundamentals and Programming", John Wiley & Sons, Inc., Toronto, 1996.
- [9] Randy Otte, Paul Patrick, Mark Roy, "Understanding CORBA (The Common Obect Request Broker Architecture)", Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1996.
- [10] Connie U. Smith, "Performance Engineering of Software Systems", Addison-Wesley Publishing Company, Inc., 1990.
- [11] D.C. Schmidt, "The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software," 12th Sun User Group, San Jose, California, June 14-17, 1993.
- [12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, " Design Patterns: Elements of Reusable Object-Oriented Software", Reading, MA, Addison-Wesley, 1994.

- [13] D.C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching", Pattern Languages of Program Design (J.O. Coplien and D.C. Schmidt, eds.), Reading, MA, Addison-Wesley, 1995.
- [14] D.C. Schmidt, "Acceptor: A Design Pattern for Passively Initializing Network Services", C++ Report, SIGS, Vol. 7, No. 8, Nov/Dec 1995.
- [15] D.C. Schmidt, "Connector: A Design Pattern for Actively Initializing Network Services", C++ Report, SIGS, Vol. 8, No.1, January 1996.
- [16] R.G. Lavender, D.C. Schmidt, "Active Object: An Object Behavioral Pattern for Concurrent Programming", Pattern Languages of Programming Conference, Monticello, Illinois, U.S.A., September 6-8, 1995.
- [17] D.C. Schmidt, "An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit", Tech. Report WUCS-95-31, Washington University, St. Louis, September 1995.
- [18] D.C. Schmidt, S. Vinoski, "Object Interconnections: Comparing Alternative Client-side Distributed Programming Techniques (Column 3)", C++ Report, SIGS C++ Report Magazine, May 1995.
- [19] D.C. Schmidt, S. Vinoski, "Object Interconnections: Comparing Alternative Server Programming Techniques (Column 4)", C++ Report, SIGS C++ Report Magazine, October 1995.
- [20] D.C. Schmidt, S. Vinoski, "Object Interconnections: Comparing Alternative Programming Techniques for Multi-threaded Servers (Column 5)", C++ Report, SIGS C++ Report Magazine, February 1996.
- [21] D.C. Schmidt, S. Vinoski, "Object Interconnections: Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers (Col. 6)", C++ Report, SIGS C++ Report Magazine, April 1996.
- [22] S. Vinoski, D.C. Schmidt, "Object Interconnections: Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers (Col. 7)", C++ Report, SIGS C++ Report Magazine, June 1996.
- [23] D.C. Schmidt, "ASX: An Object-Oriented Framework for Developing Distributed Applications", Proceedings of the 6th USENIX C++ Technical Conference, Cambridge, MA), USENIX Association, April 1994.
- [24] Hans Rohnert, "The Proxy Design Pattern Revisited", Siemen's, 1995.

- [25] Gurudas Somadder, Dorina Petriu, "Performance Measurements of Multi-Threaded Servers in a Distributed Environment", Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1997.
- [26] E. Lazowska, J. Zahorjan, G. Scott Graham, K. Sevcik, "Quantitative System Performance", Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.
- [27] Kishor S. Trivedi, "Probability and Statistics with Reliability, Queuing, and Computer Science Applications", Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.
- [28] Vikram S. Adve and Mary K. Vernon, "The Influence of Random Delays on Parallel Execution Times", Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, Volume 21, Number 1, June 1993.
- [29] Raj Jain, "The Art of Computer Systems Performance Analysis", John Wiley and Sons, Inc., Toronto, 1991.
- [30] Marshall P. Cline, "The Pros and Cons of Adopting and Applying Design Patterns in the Real World", Communications of the ACM, Vol. 39, No. 10, October 1996.
- [31] D.C. Schmidt, M. Fayad, R. Johnstone, et. al., "", Communications of the ACM, October 1996, Vol. 39, No. 10].
- [32] Dorina C. Petriu and Luminita S. Metea, "Performance Modelling and Measurement of Software Systems", IEEE Workshop on Emergent Technologies & Virtual Systems for Instrumentation and Measurements, ET & VS-IM/97, Niagara Falls, Ontario, Canada, May 15-16, 1997.

APPENDIX A

USER-SUPPLIED HEADER FILES (EODMS CLIENT)

```
// *****
//
//  Name:      Leslie Steele
//  Date:      January 23, 1997
//  Source:    application_client.h
//  Purpose:   Application include files
//             (on client side)
//
// *****
```

```
#include "dept_client.cpp"
#include "employee_client.cpp"
#include "project_client.cpp"
#include "assign_client.cpp"
```

```
// *****
//
//  Name:      Leslie Steele
//  Date:      November 16, 1996
//  Source:    config.h
//  Purpose:   Configuration file for application
//
// *****
```

```
#ifndef CONFIG_H
#define CONFIG_H
```

```
// Application-specific structures for ACE communication
```

```
struct Employee_struct    // generic structure for Employee
{
```

```

    int    emno;
    int    dpno;
    char   name[30];
};
struct Department_struct    // generic structure for Department
{
    int    dpno;
    int    mgr;
    char   name[30];
};
struct Project_struct      // generic structure for Project
{
    int    prno;
    int    mgr;
    int    hours_expended;
    char   name[30];
};
struct Assign_struct       // generic structure for Assignment
{
    int    emno;
    int    prno;
    int    hours_expended;
    char   name[30];
};
struct AI_struct
{
    int    op_code;
    int    obj_code;
    int    reply;
    char   buffer[4096];
};

#endif

```

APPENDIX B

USER-SUPPLIED HEADER FILES (EODMS SERVER)

```
// *****
//
//  Name:      Leslie Steele
//  Date:      November 16, 1996
//  Source:    application.h
//  Purpose:   Application include files
//
// *****

#ifndef APPLICATION_H
#define APPLICATION_H

#include "dept.h"
#include "employee.h"
#include "project.h"
#include "assign.h"

#define RETRIEVE_OBJ1 Employee*,Employee_struct*,int*
#define RETRIEVE_OBJ2 Department*,Department_struct*,int*
#define RETRIEVE_OBJ3 Project*,Project_struct*,int*
#define RETRIEVE_OBJ4 Assignment*,Assign_struct,int*

#define DELETE_OBJ1 Employee*,int*
#define DELETE_OBJ2 Department*,int*
#define DELETE_OBJ3 Project*,int*
#define DELETE_OBJ4 Assignment*,int*

#define ADD_OBJ1 Employee*,AI_struct*,int*
#define ADD_OBJ2 Department*,AI_struct*,int*
#define ADD_OBJ3 Project*,AI_struct*,int*
#define ADD_OBJ4 Assignment*,AI_struct*,int*

#endif
```

```

// *****
//
//   Name:      Leslie Steele
//   Date:      November 16, 1996
//   Source:    config.h
//   Purpose:   Configuration file for application
//
// *****

#ifndef CONFIG_H
#define CONFIG_H

// Application-specific structures for ACE communication

struct Employee_struct    // generic structure for Employee
{
    int    emno;
    int    dpno;
    char    name[30];
};
struct Department_struct  // generic structure for Department
{
    int    dpno;
    int    mgr;
    char    name[30];
};
struct Project_struct     // generic structure for Project
{
    int    prno;
    int    mgr;
    int    hours_expended;
    char    name[30];
};
struct Assign_struct      // generic structure for Assignment
{
    int    emno;
    int    prno;
    int    hours_expended;
    char    name[30];
};
struct AI_struct

```

```
{  
    int    op_code;  
    int    obj_code;  
    int    reply;  
    char   buffer[4096];  
};
```

```
#endif
```

APPENDIX C

BATCH SCRIPT FOR PERFORMANCE EXPERIMENTS

Gathering the performance measurements obtained for this thesis was accomplished using a UNIX batch script. The script was started from the command line in the directory in which the client and server executables were located and accepted input as follows:

`S<script> <# of clients> <experiment file> <multiplication factor>`

where

`script` = name of the main batch script

`# of clients` = maximum number of clients to be active during experiment

`experiment file` = name of file containing machines and processes to be active during the experiment

`multiplication factor` = multiplicative increment by which clients are added to the system from one execution to the next

C.1 Operation of Main Batch Script

The two batch scripts used to collect data were the main batch script and the experiment

file. The main batch script can be divided into the following major sections:

- initialization of various parameters (mean client think times, numbers of threads in server thread pool) to be used as run-time arguments by the experiment file to the server and clients
- *for* loop which for each iteration uses a different combination of mean client think time and number of threads in server thread pool to 1) remove old logging directories and create new ones; and 2) execute a *while* loop which for each desired number of clients:
 - 1) re-populates the database .NDX and .DAT files;
 - 2) uses the UNIX gawk facility to edit the experiment file to provide the command lines which launch the server and clients with the appropriate command-line arguments needed for this experiment;
 - 3) executes the experiment file once all edits of command-line arguments are completed;
 - 4) post-processes collected data once all executing processes have terminated by launching matlab to calculate the different mathematical statistics desired
- interrupt controller.

Relevant pieces of code will now be presented for each of the above sections in order to illuminate the key function(s) of that section.

C.1.1 Initialization Section

The number of threads in the server thread pool and the mean client think time are set in the following script code:

```
##Script file used for launching performance experiments
##Use the "C" shell script with the fast option
#!/bin/csh -f
unset noclobber
##On receiving an interrupt, go to CLEANUP
onintr CLEANUP

## Set the number of threads in the server thread pool
set threads = (1 4 8);

## Set mean client think time
set think = (100);

## Get executable file directory
set EXE = $cwd;
...
```

C.1.2 Experiment Loop Section

The loop commences and old logging directories are removed and new ones created in this section of script code:

```
## for each mean client think time
## for each number of threads in the server thread pool
foreach timeThink($think)
  foreach NumofThreads($threads)
    set threadsForServer = -T$NumofThreads
    set Thinktime = -t$timeThink

    #clean up old files and directories and save full experiment file
    (rm exp.old; cp $argv[2] exp.old;) >& /dev/null
    (rm -r ./expt/$NumofThreads.$timeThink;) >& /dev/null
    (rm -r ./expt/master_rsptime;) >& /dev/null
    (rm -r ./expt/master_thrput;) >& /dev/null
    (rm -r ./expt/matstats.%NumofThreads.%timeThink;) >& /dev/null
```



```
#Make and set the client experiment logging directory
mkdir ./expt/$NumofThreads.$timeThink
```

...

In the following code the *while* loop which executes for a particular number of clients, commences and re-populates the database .NDX and .DAT files:

```
while ($index <= $argv[1])

  ## Re-populate database
  (rm PERSONEL.DAT; cp PERSONEL_POP.DAT PERSONEL.DAT) >& /dev/null
  (rm PERSONEL.NDX; cp PERSONEL_POP.NDX PERSONEL.NDX) >& /dev/null

  ## Make directory for this number of clients
  (rm -r ./expt/$NumofThreads.$timeThink/client.$index;) >& /dev/null
  mkdir ./expt/$NumofThreads.$timeThink/client.$index;
```

...

The next portion of code illustrates how the UNIX gawk facility is used to edit the server command line within the experiment file:

```
## Set the number of threads in the server thread pool
gawk '{if(match($3, "server")) {print $1 " "$2 " "$3 " "$4 "varThreads" "$5 " &";"}
  else print $0}' varThreads=$threadsForServer $argv[2] >> thread.temp
## Copy the changed file to the new experiment file
rm $argv[2]; mv thread.temp $argv[2];
```

...

The gawk command above passes through each line of the experiment file and searches for a variable called *server*. Once this line is located, gawk takes the variable called *varThreads*, the number of threads in the server thread pool for this particular case, and inserts it into the appropriate place in the command line arguments for the server. A similar gawk operation occurs for setting the following command line arguments:

server logging directory (in server)
statistics directory (in client(s))
mean client think time (in client(s))
logging parameters (in client(s))

As many clients as are required to be made active for the particular run are done so as follows:

```
## Execute as many clients as indicated
@ multiplier = 1
if ($#argv == 3) then
    while ($multiplier <= $argv[3])
        gawk -f controller.awk $argv[2];
        rm $argv[2]; mv temp.exp $argv[2];
        @ multiplier ++
    end
else
    echo " $0 : Please set multiplier field"
endif
...
```

The gawk file controller.awk passes through the experiment file according to the multiplication factor in order to make the required number of clients active.

Experiment file editing has now been completed and the experiment file is ready for execution. Data processing starts with the following code:

```
## Start experiment
chmod 777 $argv[2];
$argv[2]; #launch experiment file

# Store client output file names
cd ./expt/$NumofThreads.$timeThink/client.$index
```

```

set filename = 'ls | grep client'
echo "filename = " $filename
cd ../../..

```

```

# Perform post-processing of results
foreach name ($filename)
    echo " Post processing file: ./expt/$NumofThreads.$timeThink/client.$index/$name "
    gawk '{ print $1 " "$2 >> ff } ' ff=./expt/$NumofThreads.$timeThink/client.$index/mat-
stats ./expt/$NumofThreads.$timeThink/client.$index/$name
end

```

```

echo " Finished experiment for $index clients with $NumofThreads threads and $time-
Think think time"

```

...

The above code launches the experiment file (\$argv[2] is experiment file name). The main batch file suspends execution of itself while the experiment file executes. All commands in the experiment file are executed in the background and UNIX thus requires that the experiment file and the main batch script that invoked it suspend execution until all the executing background processes

terminate. Once they terminate, control returns to the main batch script which launches matlab to calculate the statistics desired:

```

## Calculate the mean and standard deviation
matlab << monitor
clear:

wkdir = sprintf('./expt/%d.%d', $NumofThreads, $timeThink);
eval(['cd ' wkdir]);
outfile1 = fopen('matstats.stats', 'a+');

filename = sprintf('./client.%d/matstats', $index);
eval(['load ' filename]);
data = matstats(:, ...)
avg = mean(data)
...
%% Record results
fprintf(outfile1, '\t%3.6f %3.6f %3.6f\n', avg_final, tput_final, accuracy);
fclose(outfile1);

```

```
cd '../..'
monitor
```

```
## Increment index and repeat for the next combination of input parameters
...
```

The section of code between the labels "matlab" and "monitor" are designated as input commands to the matlab software.

C.1.3 Interrupt Controller

The last section of code in the main batch script relates to the interrupt controller. This controller restores the state of the experiment files in the event that an interrupt signal (ie. Ctrl-C) is received from the user.

```
##Interrupt controller
CLEANUP:
## Catch all interrupts until cleanup is complete
onintr - #ignore additional interrupts
exit 2  #exit with an error status

##Script finished
```

The "C" shell script uses the CLEANUP label to jump to that portion of code in event of an interrupt. Once in the CLEANUP code, further interrupts are disabled until experiment files and directories are restored to their original state. Afterwards, interrupts are enabled and the main batch script continues.

C.2 Operation of Experiment File

The experiment file, as mentioned above, is integral to the execution of the performance experiments controlled by the main batch script. The setup of the experiment file is shown

below:

```
## Experiment file for launching experiments
#!/bin/csh -f
unset noclobber
## On receiving an interrupt, go to CLEANUP
onintr CLEANUP

## Get executable file directory
set EXE = $cwd;

##Startup server
rsh bellatrix "$EXE/server -a54345 -Tthreads -dDIR" &;
## Allow server time to startup
sleep 30;

##Startup clients
# CLNT # rsh helicon "$EXE/client -p54345 -hbellatrix -tTHINK -dDIR -nNCYC -ILCYC
-uUCYC -rRCYC" &;
# CLNT # rsh mira "$EXE/client -p54345 -hbellatrix -tTHINK -dDIR -nNCYC -ILCYC -
uUCYC -rRCYC" &;
# CLNT # rsh sputnik "$EXE/client -p54345 -hbellatrix -tTHINK -dDIR -nNCYC -ILCYC -
uUCYC -rRCYC" &;
...
```

The expressions in capital letters above represent command-line parameters to the server and clients which are edited by the UNIX gawk facility at run-time. To launch as many clients as are required for a particular experiment, the main batch script, via gawk, strips the comment `#CLNT#` from as many lines as required. Those lines become active commands within the experiment file to launch a client at the time the experiment commences.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved